The art of active memory

Merrall, Simon C.

Award date:
1994

Awarding institution:
University of Bath

The Art of Active Memory

submitted by

SIMON C. MERRALL

for the degree of Ph.D

of the

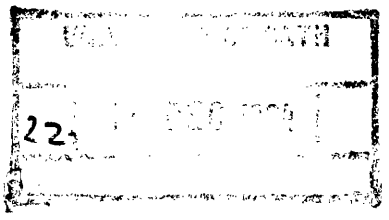
University of Bath

1994

Signature of Author.....

SIMON C. MERRALL

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346



Summary

This thesis is concerned with the design and implementation of programming languages for massively parallel architectures which reflect the *active memory* nature of such computers. We use the term active memory to describe an architecture where every storage cell has some limited processing potential. Although such computers do not as yet exist, the Connection Machine, with tens of thousands of processing elements, can certainly be viewed as a coarse-grain active memory architecture.

To identify the requirements of an *active memory programming language* we examine the ideas motivating the design of the Connection Machine. These requirements can be summarised as the ability to manipulate processors and communication with the same ease that we manipulate memory. In the same way that we create data structures using memory we should be able to use processors and communication links to create active data structures, that both represent problems and process them in parallel. A review of existing massively parallel programming languages shows that this aspect is poorly addressed.

Using Paralation Lisp, one of the better existing languages, as a basis we define extensions that allow processors to be allocated and connected to each other using an active object system. The system uses a class protocol that most lisp programmers will find familiar, but the system mechanisms have been given an active interpretation. Class instantiation corresponds to processor allocation and slot accesses to communication.

To demonstrate the ideas explored in this thesis, a fully operational implementation of Paralation EuLisp has been developed for the *MasPar* MP-1. Key elements of the implementation are discussed and illustrated to show how active objects can be realistically supported.

A selection of examples are presented showing the utility of active objects and to support our claim that active objects embody active memory programming well and give programmers a familiar and powerful interface to massively parallel architectures.

Contents

Acknowledgements

With much thanks to Keith Playford without whom a lot of this work would not have been possible. BLINDPEU and its compiler are based heavily on EUTOPIA, his reflective system for the prototyping of parallel systems. Further, his clear understanding and explanations of all things Lisp were invaluable, as were the many discussions we had about the design of TACOΣ. He also taught me everything I know about buying cheese cake.

Much thanks also to Patricia Charlton, whose friendship and company made the three and half years much more pleasant.

Thanks to Pete Broadbery for his two years off EULISP support, my supervisor Julian Padget for letting me do my own thing and donating his fridge to our office and finally to the department for giving me a part-time job after my S.E.R.C. funding ran out.

Chapter 1

Introduction

During the last decade there has been a wealth of research dedicated to the design and construction of new parallel computer architectures. As a result there are now a large number of parallel computers available in a wide variety of different processor/memory configurations, be it shared memory, processor array, multi-computer etc. This diversity can in some measure be attributed to the freedom enjoyed by the designers of computer architectures. In principal there is nothing (except funding and time) to prevent the designer exploring any avenue that may prove fruitful. If today a new architecture is to become a successful product, this will invariably be determined by the quality of its software development environment. This means an adequate programming language, preferably based on a familiar sequential language such as C or Fortran, and with the growing complexity of parallel computers a sophisticated debugger is essential. With these tools the users should find it straight forward to use the machine and in some cases they may be quite oblivious of the actual physical nature of the machine. The freedom enjoyed by the designers of computer architectures is partly a result of the insulating effect of the software development environment.

This insulating effect is advantageous in many ways, but has the undesirable side-effect of inhibiting the development of the programming languages themselves. Existing sequential languages present a well-defined interface between the programmer and the computer, so the programmer can write code for a variety of computers without difficulty. However, if the computer is parallel then a compiler must be developed that is able to detect parallelism within a program and map it onto separate processors accordingly. Although there is work being done in this field, generally vendors of new parallel architectures opt for the somewhat easier task of extending an existing sequential language, either with keywords or functions, to give access to the mechanisms supplied by the computer.

There is also a wide variety of non-vendor parallel languages that have been developed for various parallel computers. These are often parallel extensions of other sequential languages for

specific platforms, e.g. modula-2 for the *MasPar*. Others are implementations of existing parallel languages for other architectures, e.g. C*, a data parallel language developed for the Connection Machine ported to the nCUBE, once again attempting to allow a single language to be used for a variety of architectures.

Rather than simply extending existing languages with parallel constructs for each new type of parallel computer, or attempting to apply one such language to many architectures, it seems more desirable to develop languages equally suited to a wide range of parallel computers and perhaps sequential ones as well.

A great deal of work is being done in this area, attempting to identify and define high-level, canonical abstractions of parallelism, e.g. process creation, communication, synchronisation etc. The resulting languages tend to be highly abstract and much less efficient than the lower-level languages developed for specific architectures. This makes them unattractive to programmers addressing real problems where performance is a serious issue. The tendency of users to stick with the efficient languages with low-level mechanism further inhibits the development of the parallel languages because the vital feedback between designers and users is very small.

In this thesis we present work on the design and implementation of languages suitable (but not exclusively) for massively parallel architectures such as the Connection Machine (CM). The important characteristics of computers like the CM are large numbers of simple processing elements (PEs) with small local memories and excellent inter-processor communications.

The design of the Connection Machine itself is motivated by some key requirements for the simple and efficient implementation of many computationally intensive applications. The result is an architecture that can be viewed as a kind of coarse grain *active memory*. We use this term to intimate that every storage cell has some, limited, processing potential associated with it. Using active memory we can create data structures which not only represent the data, but also process the data in parallel, i.e. they are *active data structures*. Such machines do not, as yet, exist, but with tens of thousands of processing elements the Connection Machine is certainly a close relative, or rather ancestor.

The language presented here attempts to fulfil the same requirements that motivate the design of the Connection Machine. The result is an *active memory programming language* which allows active data structures to be built and operated on in parallel. The language promotes a novel programming style but doesn't need to introduce a host of new programming constructs and active data structures can be both built and processed using programming constructs familiar to many programmers.

In this chapter we will examine the Connection Machine architecture and the requirements that

motivated its design. We will see how these requirements lead to the concept of active memory and what kind of operations active memory is able to support. From this we then deduce the kind of functionality we expect from a programming language for an active memory architecture. Having formed an idea of what we expect from such a language we will then be in a position to evaluate existing languages and to design a language that meets these requirements.

1.1 The Connection Machine

In this section we give a brief description of the Connection Machine's architecture and operation. This is given purely to give the reader a background knowledge of the type of machine we are interested in, and may be found unnecessary.

The Thinking Machines Connection Machine (CM) is a massively parallel computer. It has a base configuration of 4096 processing elements and this is scalable up to 65536. The individual processing elements are very simple but this is compensated for by their sheer number. The CM also has an excellent inter-processor communication network which allows any two processing elements to communicate with each other. The CM is connected to a *host* computer which controls the operation of the processing elements. For example the host may ask each cell in a given state to add two local values and pass the result to a connected cell through the communications network. Thus a single command from the host can result in thousands of additions and a permutation of the data.

At the lowest level, the Connection Machine is a uniform array of cells, each connected by physical wires to a few of its nearest neighbours. Each cell contains a few words of memory, a very simple processor and a communicator. The communicators form a packet switched communication network allowing any cell to communicate with any other. Two cells can establish a virtual *connection* through the network which behaves as though the cells were physically connected. The Connection Machine's name refers to this ability to configure the topology of the processing elements dynamically.

The Connection Machine contains up to 64K (2^{16}) cells each with 4K (2^{12}) bits of memory and a simple serial arithmetic logic unit. The processors are connected by a packet switched network based on a Boolean n-cube topology and use an adaptive routing algorithm. All processors execute instructions from a single stream generated by a micro-controller under the direction of a conventional host. Figure 1-1 shows the basic organisation of the Connection Machine. This is a very brief overview of the Connection Machine, we now give a little more detail on the processing elements and the routers which handle communication.

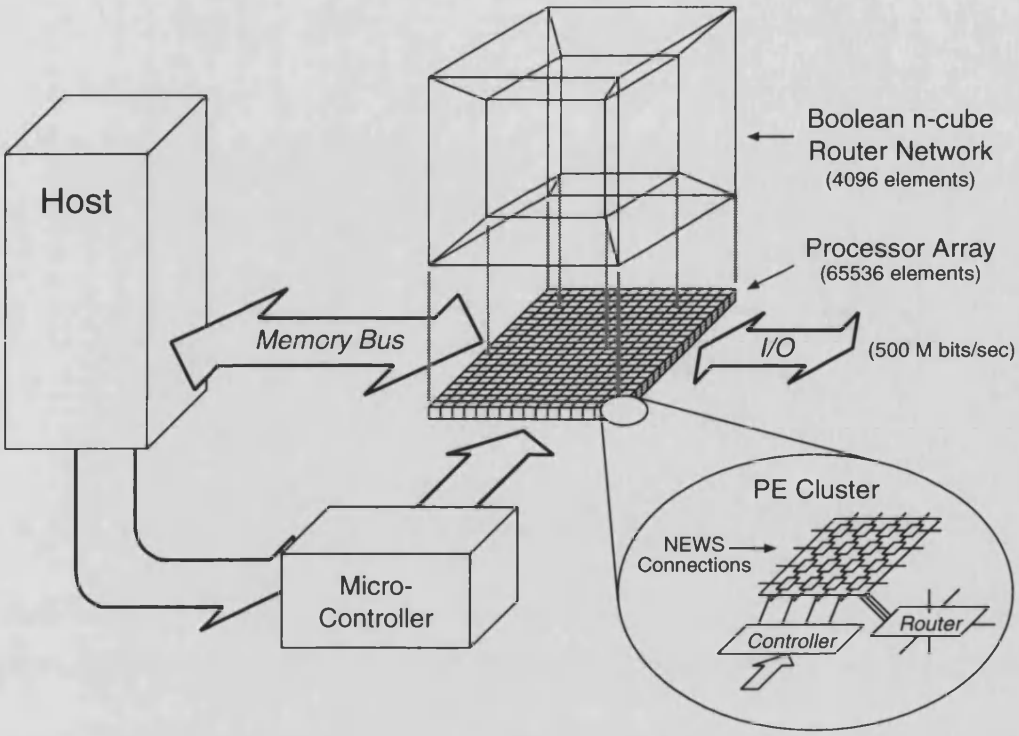


Figure 1-1: The Connection Machine

1.1.1 The Chip

The custom VLSI chip used by the CM contains 16 processor elements, a control unit and one router unit of the packet switch communications network. The control unit converts *nanoinstructions* broadcast by the micro-controller into signals controlling the operation of the processing elements and router. The individual processing elements are extremely simple, they have 8 bits of internal state information and all their paths are only one bit wide.

The basic operation of the elements is to read two bits from local memory and an internal flag, combine them producing a 2-bit result and write one bit to memory and the other to an internal flag. All the parameters for this operation are specified in a single *RISC* style instruction. The processors all receive the same instruction stream from the control unit, but the processors conditionally execute each instruction depending on the state of one of the processor's internal flags. The CONDITION-FLAG parameter specifies which flag is to be used for this purpose, and the CONDITION-SENSE parameter specifies how the flag is to be interpreted.

The processors are connected in a 4×4 grid allowing each processor to communicate directly with its North, East, West and South neighbours. This two dimensional grid is extended across multiple chips by connecting the *NEWS pins* of adjacent chips.

1.1.2 Router Communication

Each router handles messages for the 16 processing elements on its chip. Thus the communications network for a 64K Connection Machine contains 4096 routers. The routers are wired in the pattern of a Boolean n-cube; in a network of n nodes with this topology the distance between any two nodes is $\leq log_2 n$.

Geometrically, the Boolean n-cube can be interpreted as a generalisation of a cube to an n-dimensional Euclidean space. Each router has a 12-bit address which gives its position in the Boolean n-cube. There is one bit in the router address for each dimension of the the Boolean n-cube. An edge of the cube pointing along dimension k connects two vertices whose addresses differ in the kth bit. As any two 12-bit addresses differ by no more than 12 bits, each router can be no more than 12 wires away from any other router.

Each communication cycle is made up of 12 *dimension cycles*, one for each dimension of the hyper-cube. During the cycle messages are moved across each of the 12 dimensions in sequence. In a Boolean n-cube a message can be no more than one step away from its destination per dimension; thus all messages are delivered within one communication cycle, unless they are delayed by traffic.

1.2 Why Build a Connection Machine?

Most computers have a two part design where the memory is separate from the processors, this is known as the von Neumann architecture. This division was originally made for good reasons, processors consisted of relatively fast and expensive switching components such as vacuum tubes where as memory was made from relatively slow and relatively inexpensive components like delay lines or storage tubes. This basic design has been so successful that designers have kept using it even though the technical reasons for it no longer exist, today both processors and memory are made of the same material, i.e. silicon.

In a modern von Neumann computer almost all the transistors are devoted to memory. This means that at any time only a few of the transistors in the computer are active, those within the processors and any memory being accessed. The memory/processor division keeps the silicon devoted to processing as busy as possible, but this is only 2 or 3 percent of the total silicon, the remaining 97 percent remains idle. This seems an expensive resource to be wasting in this way.

As machines become bigger and bigger the problem gets worse, memory scales easily but processors do not. As a result the ratio between memory and processors gets larger giving greater inefficiency. The inefficiency remains no matter how fast we make the processor because any

computation becomes dominated by the time required to move data from memory to the processor. This is known as the von Neumann bottleneck.

1.2.1 Concurrency Offers a Solution

An answer to the problem is to scrap the von Neumann architecture and build a homogeneous computing machine where memory and processing are combined. This way a higher percentage of the silicon will be kept active giving us more processing power per square metre of silicon. Although we can build machines like this it is not immediately obvious that we can use them. How does one decompose an application into thousands of parts that can be executed concurrently? And how does one then coordinate those tasks to produce the final result?

There are reasons to believe that calculations can be performed with such a high degree of concurrency. We have the example of the brain which efficiently solves complex problems with apparently slow switching components. Cellular automata [45] are able to model globally complex systems with large numbers of locally simple processes. There are also various examples where high degrees of concurrency can be achieved by matching processing elements to the natural structure of the data. Image processing, VLSI simulation and semantic networks are a few such examples.

1.2.2 The Requirements for a Connection Machine

In his thesis, Hillis derives the requirements for a Connection Machine by examining a particular parallel algorithm; finding the shortest path between two vertices in a large graph [30, pp. 10–20]:

> Given a graph with vertices V and edges $E \subset V \times V$, with an arbitrary pair of vertices $a, b \in V$, find the length k of the shortest sequence of connected vertices $a, v_1, v_2, \ldots b$ such that all edges $(a, v_1), (v_1, v_2), \ldots (v_{k-1}, b) \in E$ are in the graph.

The algorithm for finding the shortest path from vertex A to vertex B begins by labelling every vertex with its distance from A. This is accomplished by labelling vertex A with 0, labelling all the vertices connected to A with 1, labelling all unlabelled vertices connected to them with 2, and so on. The process terminates as soon as vertex B is labelled. The label of B is then the length of the shortest connecting path.

Algorithm I: *Finding the length of the shortest path from A to B*

1. Label all vertices with $+\infty$

2. Label vertex A with 0
3. Label every vertex except A, with 1 plus the minimum of its neighbours labels and itself. Repeat until label of B is finite (not ∞)
4. Terminate. The label of B is the answer.

Algorithms of this type are slow on conventional machines. Assuming that each step takes unit time then the algorithm terminates in time proportional to the length of the connecting path. Unfortunately the steps in the algorithm do not correspond well to those executed by a von Neumann machine. Direct translation of the algorithm gives a programs that terminates in time proportional to the number of vertices times the length of the path times the average degree of each vertex.

Another disadvantage of a serial implementation is that as well as iterating over the algorithm steps it also has to iterate over the vertex set. As a result we immediately move a step away from the algorithm making the program harder to understand. In addition most good programmers would automatically add various optimisations to the code making the the program still harder to understand. Further optimisations often *tune* a general algorithm for a specific subset of examples. In general optimisations trade speed for clarity and flexibility.

Rather than optimising the algorithm to match the architecture we could make a machine which matches the algorithm.

Requirement I: Many Processors

The algorithm describes steps which operate on entire sets of vertices simultaneously, so in order to implement the algorithm directly we will need concurrency. To perform an operation on each vertex of the graph concurrently we will need a separate processing element associated with each vertex.

This of course means we need to be able to supply an arbitrarily large number of processors. Though we clearly cannot do this we can build a machine with sufficient processors to meet the requirements of most applications. We are used to similar restrictions with memory on conventional machines where we assume there is sufficient memory for our needs but recognise there is a finite limit.

A corollary to this requirement is that each processing element is as small and as simple as possible so that we can afford to have many of them.

Requirement II: Programmable Connections

In the path length algorithm, the pattern of inter-element communication depends on the structure of the graph. The machine must work for arbitrary graphs, so every processing element must have

the potential of communicating with every other processing element. In addition, for some problems we may wish to change the communication pattern during a computation, so the inter-element connectivity must be part of the changeable state of the machine.

Although we require programmable connections the processors themselves may be connected by fixed physical wires. This means that communication will be easier for some processors than others. A similar situation occurs with virtual memory where accessing a resident memory page is faster than accessing a page held on disk, but this is hidden from the software which considers accesses to the two locations to be of equal cost. In the same way the physical locality of memory is hidden in a von Neumann machine we would like our machine to hide the connectivity of its processors.

1.2.3 The Connection Machine Architecture

The two requirements identified in the previous section can be summarised as:

Requirement I : Enough processing elements to be allocated as needed in proportion to the size of the problem.

Requirement II : The processing elements can be connected by software.

The Connection Machine architecture directly follows from these two requirements. It contains a large number of simple processor/memory cells connected by a programmable communications network. The Connection Machine is connected to a host computer which builds *active data structures* on the Connection Machine in much the same way they are stored in conventional memory. The host then controls the activities of these structures specifying local computation and inter-processor communication.

1.2.4 Active Memory

In sections 1.2.2 and 1.2.3 we identified two requirements that must be satisfied by a parallel system if certain kinds of computationally intensive task are to be supported effectively. These requirements are not in themselves particularly novel as modern day memory satisfies precisely these same requirements; we are so used to working with conventional memory however that we probably do not view it in this way. If we consider the process of building a conventional data structure we can see that memory must also satisfy requirements I and II.

To build a data structure we allocate memory segments as and when they are needed. This corresponds to the first requirement: *many processors*, i.e. there is sufficient memory to meet our

needs. By connecting memory segments to each other using pointers we are able to build any desired data structure. This corresponds to the second requirement: *programmable connections*, i.e. memory segments are *connected* using their addresses as pointers. So we could paraphrase the Connection Machine's requirements as:

> Processing elements and communication links can be allocated and manipulated with the same ease as memory.

Thus these requirements define an architecture that can be thought of as a kind of *active memory*. Taking the requirements to their limit we can envisage a computer where the processing elements have become so fine-grained that they are equivalent to a single word in a conventional computer. This would give us a computer where every storage cell had some limited processing potential and was able to read the contents of any other cell within the computer, a truly *active* memory. Data structures could be built in active memory in the same way as in conventional memory, these structures would process as well as represent the data and also be able to dynamically reconfigure themselves.

A computer supplying such fine-grain concurrency is, no doubt, an impossible objective. But the Connection Machine, with tens of thousands of processors, is definitely a coarse relative of this fabled machine. The Connection Machine is connected to a conventional computer much like a conventional memory and its internal state can be read and written a word at a time from the conventional machine. It can be used to build data structures in the same way as conventional memory, and these structures *do* both represent and process the data.

We use the term active memory to intimate this aspect of the Connection Machine. Other systems have also used this term, for example, in an implementation of the Subset Abstract Machine (SAM) [64] for the CM2, the component responsible for storing and operating on sets in parallel is described as an active memory. This is a weaker use of the term as it only refers to the ability of the storage medium to process its entries. This is because systems like SAM are oriented around simple collection data structure like sets and *bags* [37]. We are interested in more complex data structures, like trees and graphs, and as such our active memory needs the additional property of any cell being able to access the contents of any other cell by its address.

1.3 Programming Active Memory

Having shown how massively parallel computers like the Connection Machine represent a coarse grain active memory, the next question to consider is how these computers are programmed.

The simple approach to defining a programming language for these computers is to produce a language which gives access to all the mechanisms supplied by the architecture. Which is to say, data can be stored on a set of processors and then operated on in parallel, with processors participating in the computation conditional on some activity flag. The computation can also include inter-processor communication, where every processor reads or writes a value from or to another processor specified by its address/identifier. This is in fact what the bulk of the massively parallel programming languages do, good examples being *Lisp and mpl, which give very precise control over the Connection Machine and *MasPar* respectively. But although these languages give excellent control over the machines, i.e. there are no features of the architecture that cannot be utilised by the programmer, they do not embody the active memory nature of the computers well. Rather than building data structures as we would with a conventional computer, collections must be created specifying the communication patterns that correspond to the desired structure.

A more interesting approach is to define an active memory programming language. This will allow the processors of a massively parallel computer to be manipulated in much the same way as conventional memory. The processors will be used to create active data structures much like conventional data structures. Then rather than having a process traverse the structure, propagating data and performing local computations, a process can execute on each processor in the structure in parallel, performing local computations and moving data between the processes via the communication links in the structure.

As well as being interesting in its own right, there are also some perceived advantages of such a language:

- Programmers are used to working with memory, and so will find massively parallel computers easier to use if they have the appearance of active memory.
- Being able to create, manipulate and use active data structures directly will eliminate the need for devising collection based representations and converting to this representation.
- The language mechanisms to manipulate the processors and communication links will be similar to those handling memory. This will help reduce the amount of new and unfamiliar mechanisms needed in the language.

This is the objective of the work presented in this thesis, to define a language for massively parallel architectures that embodies their active memory nature. The work extends an existing language, Paralation Lisp – a high-level, architecture independent, parallel programming language – with an *active object system*, TACOΣ. This uses the ideas of object systems to manage processors and

communication; as a result the mechanisms supplied will be familiar to many programmers. As well as fulfilling the requirements of an active memory programming language, the use of object-oriented technology to capture the active memory nature of fine-grained parallelism also opens up further opportunities for using object systems to capture other aspects of parallelism.

1.4 The Rest of the Thesis

Although this work is essentially language independent it is presented here in the context of EuLisp, this lisp dialect being the main platform for parallel language research currently in progress at Bath University. As well as being an ideal platform for language design and development, the existing data-parallel languages of interest are also lisp based. A working knowledge of lisp is assumed throughout the thesis, but familiarity with EuLisp itself should not be necessary. In Chapter 2 we review the existing massively parallel functional languages and consider how well they fulfil the requirements of an active memory programming language. The rest of the thesis can be split into three parts.

1. **Definition:** Having identified the requirements for the language here and in Chapter 2 we go on to look at extensions to Paralation Lisp and consider how they meet these requirements. From these extensions we make various useful observations which motivate some key aspects of the active object system's (TACOΣ) design, which is presented in Chapter 3.

2. **Usage:** In Chapter 4 we experiment with the object system to discover if it enhances the base language. Various interesting mechanisms naturally supported by active memory programming are examined and some alternative language syntax etc. is also discussed.

3. **Implementation Issues:** Having presented the functionality of the new language it is important that it can be realistically implemented. The language raises several implementation problems which are discussed in chapters 5 and 6.

We finish by looking at some related work, and considering how both the design and implementation of the active object system may be improved. We also consider how other object-oriented mechanisms could be added to the system to enhance it in general and to also capture other interesting aspects of parallelism.

Chapter 2

Reviewing the Language Barrier

We have now characterised the class of computer architecture we are interested in, the so called *active memory* computers, and established some requirements for, or at least expectations of, their programming languages. We will now look at some of the languages which have been developed for these machines and see how well they meet our requirements. There is, in fact, a very large number of these languages, most of which were developed (initially at least) for some SIMD platform. That is a parallel computer (like the CM) where each processor executes the same instruction stream, hence Single Instruction Multiple Data as opposed to Multiple Instruction Multiple Data (MIMD) where each processor executes its own instruction stream. Most of these machines have at least one vendor supplied language specifically designed for that machine, usually extended versions of C [38, 53] or Fortran [39, 44]. More recently, some of these languages have been made available for other machines or indeed architectures, An example of this is C*, originally developed for the CM-2 it is available for the CM-5 which supports both SIMD and MIMD execution models and has also been implemented for some multi-computers like the nCUBE 3200 and the Intel iPSC/2[28]. There are also many independently developed languages, for example there are several data parallel dialects of Modula-2 [12, 20, 48].

Most of these languages supply some sequence data type whose elements can be operated on uniformly in parallel, i.e. the same operation may be applied to each element in parallel. This may be a specific type, like the multi-dimensional arrays in Fortran, or extra syntax is supplied to specify when a variable should be instantiated in parallel. The language constructs effecting inter-processor communication usually reflect the mechanisms in the development platform closely but some have more abstract communication operations, e.g. matrix transposition. There are also some limited facilities for defining the size of a processor set to be used and its topology.

Although the work in the field of data-parallel procedural languages is important, our interest lies

with the functional and applicative style languages such as Scheme, Lisp and ML at which we will now take a more detailed look.

2.1 Functional Data Parallel Languages

We give here a basic outline of some of the key functional and applicative data parallel languages. This section aims to make the reader familiar with how parallel programmes are written with these languages, general terminology and to draw attention to some notable features of data parallel programmes. We defer a critique of their various merits until the next section.

2.1.1 *Lisp

*Lisp [66] is an extended version of Common Lisp [60] developed for programming the Connection Machine. It supplies a very large number of functions which give the programmer complete control over the processor array. For some time it was the main development language for the CM-2, being more efficient than Connection Machine Lisp (*c.f.* section 2.1.4), predating C* and easier to use than ParIS[67] the CM-2's parallel instruction set.

*Lisp supplies a new sequence data structure called a *pvar* (short for parallel variable), which is similar to a vector where each element is stored on a separate processor. We use the general term *data parallel object* to describe collections of objects like pvars which can be operated on in parallel. It should be mentioned that the CM-2 has a virtual processor mechanism which operates at the instruction level. It makes more virtual processors available by repeatedly dividing the memory of the physical processors in half. So within the limits of memory, programmers may specify the number of processors they wish to use. In this way pvar size can be varied, but it must be a power of two of the physical number of processors.

*Lisp is designed to allow the programmer to get the best possible performance out of the Connection Machine. To this end the type of a pvar can be declared, this constitutes a promise to the compiler about the contents of that pvar allowing more efficient code to be generated.

*Lisp provides parallel versions of most serial Common Lisp operators. They are distinguished from their serial counterparts by a `!!` suffix, e.g. parallel addition is `+!!` and `=!!` is the parallel equality predicate. Flow of control and other syntactic operators like `if` and `let` are distinguished by a `*` prefix. In addition, the unary operator `!!` projects a singular value into a pvar, i.e. it returns a pvar containing its argument in each element. To give an idea of the format of *Lisp, below is some code to count the number of non-nil elements in a pvar of variable length one dimensional arrays.

```
(defun counts (arrays idxs lens)
  (*if (<!! idxs lens)
    (+!! (if (*or (!! t)) (counts arrays (!!+-1 idxs) lens)
        (!! 0))
      (*if (null!! (aref!! arrays idxs)) (!! 1)))
    (!! 0)))
```

This example also highlights an important aspect of data-parallel programs. To execute a conditional form on a SIMD computer we first evaluate the boolean decision expression and activate only those processors for which it is true, we then execute the consequent code. After this we activate all the processors for which the boolean was false and execute the alternative code, the two sets of results are then combined into a single parallel result. This means that in general both the consequent and the alternative code will be executed.

In the `*if` form in the example above the consequent code contains a recursive call which would naively always be evaluated even if there were no processors active. Thus the function `counts` would recurse until it ran out of stack space and then fail. In order to prevent this the call to `counts` is wrapped by a singular conditional form which only evaluates the consequent form if there are any active processors. It does this by projecting `t` into all the active processors and applying `*or` to the resulting pvar to determine if any of its elements are non-nil.

The Connection Machine has two communication networks, the nearest neighbour NEWS network and the boolean hyper-cube router network. *Lisp has sets of functions for accessing both these mechanisms. The basic *Lisp functions for regular communication are `news!!` and `*news`. They are used to shift data uniformly across grids of any dimension, although they are most commonly used for two-dimensional grids. The expression below will shift the grid of values in `source` pvar up one and left one.

```
(news!! source 1 1)
```

Irregular communication is effected by the functions `*pset` and `*pref`. With `*pref` each active processor reads a value from a specified processor which need not be active. When using `*pset` it is possible that collisions will occur in the destination processors in which case the user can specify how they are to be combined. The example below writes the contents of `pvar1` into `pvar2` in reverse order (note `self-address!!` returns a pvar of each processor's address and in this case we know there are no collisions).

```
(*set pvar1 (self-address!!))

(*pset :no-collisions pvar2 pvar1
       (-!! *number-of-processors-limit* (self-address!!)))
```

Finally `*or` is a member of a set of reduction operators which reduce a pvar by some associative operator, other examples are `*max` and `*and`.

2.1.2 TUPLE

TUPLE [72] is a data parallel version of Kyoto Common Lisp developed on the *MasPar* MP-1 at Toyoyhashi University, Japan. Like *Lisp, TUPLE has a relatively low-level abstract data-parallel model. But not being developed to give *absolute* control over the processor array it is much less primitive. It is still a very efficient implementation and is currently the best version of lisp available for the *MasPar* (see Section 2.2.2).

Although *Lisp and TUPLE are both efficient and relatively low-level lisp languages they are functionally quite different. In *Lisp a parallel expression is simply an ordinary lisp expression which contains parallel functions and forms so that the expression manipulates data-parallel lisp objects. In TUPLE the data-parallel component is disjoint from the serial part of the system. The programmer defines parallel functions, variables etc. and invokes parallel execution using special forms. This form is executed entirely in parallel to completion and then a result is returned. In this way TUPLE is like an ordinary lisp process which has a separate data-parallel lisp system embedded in it which is accessed through a relatively compact set of functions and special forms.

Below we define a parallel function and use it to create a parallel variable of descending lists of integers:

```
(defpefun en-to-one (n)
  (if (> n 0) (cons n (en-to-one (- n 1))) ()))

(ppe (en-to-one (rem penumber 5)))
⇒ #P(() (1) (1 2) (1 2 3) (1 2 3 4) ... )
```

It is interesting to note how this different model of data parallel execution neatly side-steps the problem of singular side effects in data-parallel expressions at the language level. The programmer need only consider the *micro* (*c.f.* section 2.3.2) when writing parallel code. If the code will behave correctly on a single processor then it will do so when executed in parallel on a large set of processors.

The lisp programmer is able to leave the task of generating code correct for data parallel execution to the compiler. However so that the programmer is not restricted to a programming model of independent parallel processes TUPLE provides the special conditional form `exif`. This is used in the same way as `if`. However if the consequent is executed by any of the PEs then the remaining PEs simply return `nil` – so whereas an `if` form is executed independently on each PE, the execution of `exif` is determined by the state of *all* the PEs.

TUPLE is strongly geared towards parallel list processing, the only objects allocated on the PEs are cons cells (called pons cells), all non-immediate data is allocated on the host computer making their use very slow. This system still has certain advantages and makes it possible to give the MASPAR a uniform address space. As a result, in TUPLE, both the processing elements and the front-end can address an object on any other processor.

As with *Lisp there are functions for each of the different inter-processor communication operations of the MASPAR. There is a set of functions for the 8-way nearest neighbour communication network, e.g:

(`mgetn` obj_1 [obj_2])

And a single function for router based communication.

(`gget` obj_1 *fix* [obj_2])

Communication takes place on all those processors currently active, obj_1 is the value that each PE sends, obj_2 is a default value if the specified PE is inactive. We can think of this as each PE making a value available to be read by other PEs and then itself attempting to read a value. There is no counterpart to the write operations in *Lisp, also in *Lisp the PEs are not restricted to communicating with the currently active processors.

TUPLE also has a good selection of reduction operators like *Lisp, some examples are `reduce-+`, `reduce-min` and `not-every-pe`.

2.1.3 **Plural** EULISP

Plural EULISP [42, 40] is an experimental, data parallel extension to EULISP[46] developed at Bath University for the MASPAR MP-1. It is worth a brief mention because although fairly low-level it has abstractions of processor management and communication. EULISP itself is a parallel dialect of Lisp developed at Bath and in conjunction with academic and industrial researchers around Europe. The distinguishing features of the language are modules for separate compilation, threads for multi-tasking and a fully integrated object system based on classes and generic functions. A more detailed description can be found in the language definition [47].

Plural EuLisp supplies a new sequence data structure called a *plural* which again is similar to a vector where each element is allocated on a separate processor. Unlike the data parallel objects in *Lisp and TUPLE the size of a plural is not that of the physical array. A plural is created using the function `make-plural` which takes the desired length of the plural as its argument. For example:

```
(setq a (make-plural 5))
⇒ #P(() () () () ())
```

The initial value of each element of the plural is `nil` (the empty list). We can set and reference elements of the plural using the function `plural-ref` and its updator:

```
((setter plural-ref) a 1 '(1 a))
⇒ #P(() (1 a) () () ())
```

Plural EuLisp has a set of primitive functions which can be applied to plurals. These are data parallel versions of typical lisp primitives. They are usually distinguished by a `-s` suffix (e.g. `car-s`, `null-s`), but where there is an appropriate generic function the data parallel version has been added as a method (e.g. `+`). When the function is applied to a plural it is as though the serial version of the function were applied to each value in the plural and the result is a new plural containing these individual results.

```
(null-s a)
⇒ #P(t () t t t)
```

The values in the resulting plural are allocated on the same set of processing sites as the argument plural as this is where they were created. In this case the two plurals are said to be *conformant* or to belong to the same *conformant set.*

There is an additional data parallel function, `bang`, which has no serial counterpart. This projects a singular value into a plural, for example:

```
(setq b (bang 55 a))
⇒ #P(55 55 55 55 55)
```

This creates a new plural, conformant to `a`, with each element set to `55`. If a data parallel function takes more than one argument (e.g. `cons-s`) then they must be conformant. So

```
(cons-s b a)
⇒ #P((55) (55 1 a) (55) (55) (55))
```

is correct because `a` and `b` are conformant, but `cons-s b (make-plural 5)` signals an error as the new plural would not be conformant to `b`. To make it easier to allocate conformant plurals,

the argument to `make-plural` can be a plural instead of an integer and in this case the resulting plural is conformant with the argument. Similarly the conversion functions `list-to-plural` and `vector-to-plural` accept an optional plural argument to which the result will be conformant—padding or truncating the list or vector data as necessary.

The parallel conditional form in Plural EuLisp differs slightly from those in the previous languages. The arguments to `if-s` are three expressions which deliver conformant plural values. As in the other languages the alternative or consequent form is executed on each processor depending on the value of the boolean expression, so in this way it is similar to `*if` in *Lisp. However if there are no processors active for one of the expressions then it is not executed at all, making it behave more like `if` in TUPLE.

This feature of Plural EuLisp makes writing parallel lisp programs more natural as it eliminates the problems with singular side effects in parallel expressions described earlier. For example the intuitive definition of parallel list length below behaves correctly, but without this property of `if-s` it would have recursed until an error occurred:

```
(defun list-length-s (list-s)
  (if-s list-s (+ (bang 1 list-s) (list-length-s (cdr-s list-s)))
    (bang 0 list-s)))
```

Plural EuLisp has an high-level abstraction of communication which is based closely on that in Paralation Lisp and is described in full in section 2.1.5.

2.1.4 Connection Machine Lisp

Connection Machine Lisp[61] is a highly abstract data parallel version of Common Lisp with a strong algebraic feel to it. The data parallel objects are *xappings*[1], which have three components: a *domain*, a *range* and a *mapping* between them. Xappings can be represented as an unordered set of ordered pairs *index* → *value*, where the *index* is a member of the xapping domain and the *value* is the member of the range to which it is mapped, e.g:

```
{sky→blue grass→green apple→red}
```

A xapping where each element of the domain is mapped to itself is called a *xet* and has a special representation:

```
{a→a 1→1 2→2} ≡ {a 1 2}
```

[1] The description here is based on [61] which differs slightly from that given in [30]

Another special case is where the domain of the xapping is a contiguous sequence of integers starting at zero, this is called a *xector* and also has its own representation:

```
{0→a 1→b 2→c} ≡ [a b c]
```

The final special case is a constant xapping where all elements in the domain are mapped to the same value. In this case only the range is written:

```
{ →3}
```

The elements of a xappings range can be referenced using the function `xref` by specifying the corresponding value in the domain of the xapping, `xset` can be used to update an element of a xapping. If the value is not in the domain an error is signalled. The function `xmod` behaves like `xset` but if the value is not in the domain it adds a new index/value pair to the xapping.

Connection Machine Lisp supplies the *alpha* (α) operator to convert a single value into a constant xapping. Alpha can be applied to a function to create a xapping of functions and this is how parallel computation is expressed in CM-Lisp. A xapping of functions can be applied to xapping arguments, in which case the function is applied concurrently to the elements of the argument xappings with corresponding indices. The result is a xapping of the individual results.

```
(α+ '{a→1 b→2} '{a→3 b→3}) ⇒ {a→4 b→5}

(αcons '{a→1 b→2 c→} '{a→3 b→3}) ⇒ {a→(1 .  3) b→(2 .  3)}

(αcons '{a→1 b→2} α9) ⇒ {a→(1 .  9) b→(2 .  9)}
```

Alpha distributes over the expressions it is applied to so the two expressions below are equivalent:

```
α(+ 1 2) ≡ (α+ α1 α2)
```

This is a very useful property as it makes parallel expressions much simpler but α can only be applied to expressions which have no parallel sub-expressions. This makes the ability to factor α out of expressions next to useless as we simply get some constant xapping as a result. To remedy this CM-Lisp supplies another operator, •, which cancels the affect of α. This is similar to the Common Lisp backquote notation. Backquote can be thought of as "make a copy of the following data structure" and comma as "but don't copy this, use its value instead". In the same way α can be thought of as "perform multiple copies of this expression in parallel" and • means "but don't copy this, use elements of its value (which is parallel)"

```
α(+ (* •x 2) 1) ≡ (α+ (α* x α2) α1)
```

Communication in Connection Machine Lisp is effected by the β operator. Beta has two modes of operation, the first and simpler is as a reduction operator. Beta takes a binary function and gives a

function which reduces a xapping to a single value by combining the values using the binary function. For example:

```
(β+ '{a→1 b→2 c→3}) ⇒ 6
```

The second, more general, form of β takes as arguments a combining function and two xappings. It returns a new xapping where the values are those of the first xapping and the indices are specified by the values of the second xapping.

```
(β '{A→1 B→2} '{A→X B→Y}) ⇒ {X→1 Y→2}
```

Operationally this can be understood as the values of the first xapping being sent to the processors which have the labels specified by the second xapping. If the value of the xapping specifying the indices contains any repetitions then more than one value will be sent. In this case, the corresponding values in the range xapping are combined using the given binary function. In this way β serves as a general inter-processor communication form.

```
(β+ '[1 2 5] '[x z z]) ⇒ {x→1 z→7}
```

2.1.5 Paralation Lisp

The Paralation Model is a high-level, architecture independent, parallel programming language devised by Gary Sabot [55], the description given here is based on the functionality of Paralation Lisp.

The model adds a new sequence data structure to the base language called a *field.* A *paralation* (a contraction of "parallel relation") is a set of related fields. Informally a paralation is a set of sites and a field a collection of objects, one for each site in the paralation. Each paralation has a unique member called the *index* field which enumerates the sites of the paralation from 0 to $(n-1)$. The fields in a paralation have *element-wise locality* which means that the ith elements of all the fields in a paralation are *near* each other.

The function `make-paralation` allocates a new set of processing sites and the index field for the new paralation is the return value.

```
(setq p (make-paralation 5))
⇒ #F(0 1 2 3 4)
```

The function `index` can be applied to a field to find the index field of its paralation. Because the index field is a unique member of the paralation this gives a simple test for determining whether two fields belong to the same paralation.

```
(eq (index p) p)
⇒ t
```

Parallel computation is expressed using the `elwise` form. This takes a list of identifiers bound to fields (in the same paralation) and some lisp expression. The expression is evaluated in parallel on each of the sites in the paralation. On the sites the identifiers are bound to the element of the field in that site rather than the entire field. In effect `elwise` is a parallel `let` form and indeed `elwise` can make local bindings in the same way that `let` does. The result is a new field in the same paralation. The value of each element is the result of executing the expression on that site of the paralation. Below we convert a list to a field and use the temporary binding `elt`:

```
(setq from (elwise ((elt p)) (list-ref '(nowhere 1st 1st 2nd nowhere) elt))
⇒ #F(nowhere 1st 1st 2nd nowhere)
```

The paralation model provides an abstraction of inter-processor communication called *mappings*. Informally a mapping can be thought of as a bundle of one-way arrows connecting sites in a source paralation to sites in a destination paralation. Given two fields the function `match` creates a mapping connecting the sites in the two paralations that have equal values in the argument fields.

```
(setq map (match (elwise ((to (make-paralation 3)))
                         (list-ref '(1st 2nd 3rd) to))
                 from))
⇒ #<mapping>
```

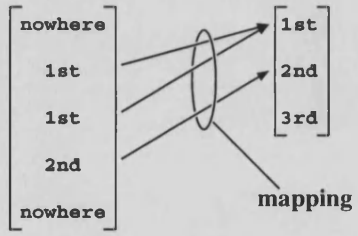


Figure 2-1: Creating a mapping using `match`

The function `move` allows us to move a field in the source paralation of a mapping to the destination paralation, each value in the source moves down the mapping *arrows* to sites in the destination paralation to create a new field. It is possible that there will be no arrows pointing to a site in the destination and a default value is supplied for this case. If more than one arrow points to a site then a collision will occur and a given binary function is used to combine the colliding values.

```
(setq data (elwise (p) (list-ref '(a b c d e) p)))
⇒ #F(a b c d e)
(move data map cons 'empty)
⇒ #F((b . c) d empty)
```

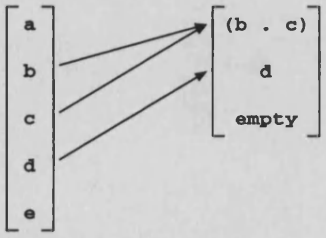


Figure 2-2: Moving data between paralations

In figure 2-2 a collision between two objects occurs in the first element and they are combined into a dotted pair. The last element has no counterpart in the source and takes the default value `empty`.

Paralation lisp also has a general reduction operator which behaves very much like the simple monadic version of β in Connection Machine Lisp. The function `vref` reduces a given field to a single value by combining the field values using a given binary function:

```
(vref p +)
⇒ 10
```

2.1.6 NESL

NESL is a strongly typed, applicative, data parallel language with an ML-like [26] syntax devised by Guy Blelloch [7]. Parallelism is supplied through a simple set of data parallel constructs based on sequences. As well as a broad set of parallel functions which manipulate sequences there is a mechanism for applying any function over all the elements of a sequence in parallel.

The application of a function to the elements of a sequence is specified using a set like notation similar to *set-formers* in SETL [58] and the *list-comprehensions* of Haskell [32] and Miranda [68]. Below the set notation is used to create a sub-selection of the sequence `[7, -2, 5, 4]` and then apply the function `negate` to each element of the resulting sequence.

```
{negate(a) : a in [7, -2, 5, 4] | a < 5};
⇒ [2, -4] : [int]
```

Where paralation lisp has the `vref` operation NESL supplies a set of reduction operators like `sum`, it also supplies a set of scan operators:

```
sum([2, 1, -3, 11, 5]);
⇒ 16 : int


plus_scan([1, 3, 5, 7, 9, 11, 13, 15]);
⇒ [0, 1, 4, 9, 16, 25, 36, 49] : [int]
```

There is a large selection of functions which manipulate the elements of vectors which are used for doing inter-site communication:

```
permute("road", [2, 1, 3, 0]);
⇒ "dora" : [char]
```

NESL also supplies as primitives two very important functions which make it possible to move between levels of nesting in nested vectors. The efficient use of nested vectors is an important issue

in data parallel languages (see Section 5.3) and Paralation Lisp has similar (though not primitive) functions (see Section 2.5). The functions are `flatten` and `partition`:

`values`	=	$[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$
`counts`	=	[4, 1, 3]
`(partition values counts)`	=	$[[a_0, a_1, a_2, a_3], [a_4], [a_5, a_6, a_7]]$
`values`	=	$[[a_0, a_1, a_2], [a_3, a_4], [a_5, a_6, a_7]]$
`(flatten values)`	=	$[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$

One of the key goals of the language design is that the asymptotic complexity can always be derived from the code as a function of the length of the vectors used in the code. For this reason NESL has no high-level, abstract and powerful operators like β in Connection Machine Lisp but instead a set of orthogonal functions with well-defined cost functions. There are two complexities associated with all computations in NESL.

1. **Work complexity:** this represents the total work done by the computation, that is the amount of time the expression would take if executed on a serial random access machine. This is usually the size of the vectors being operated on.

2. **Step Complexity:** this represents the parallel depth of the computation, that is the amount of time the expression would take if executed on a machine with unlimited processors. The step complexity of all NESL functions is one.

These complexities are based on the vector random access machine (VRAM) model [6] which is a strictly data-parallel abstraction of the parallel random access machine (PRAM) model [34]. Many of the step complexities are derived from the argument that many logarithmic time operations can in fact be considered as unit time operations [5].

In some ways NESL is a version of Paralation Lisp that has been greatly cut down to improve performance. The vectors in NESL are similar to typed fields, the strong typing gives regular programs making it easier for the compiler to generate more efficient code, this is discussed later in section 5.3. The `over` form is very similar to `elwise` and much of the functionality of `vref` and mappings is supplied by the numerous, simple and efficient vector manipulating functions of NESL.

2.2 A Critique of the Low Level Languages

Of the languages we looked at in the previous section, *Lisp, TUPLE and Plural EULISP, were all relatively low-level languages in which no real attempt had been made to abstract the mechanisms of

data-parallelism. Although this means they are not of great interest to us in themselves, they are still worth examining as they give us a background for examining the more abstract languages. *Lisp and TUPLE are of additional interest as they closely reflect their development platforms, the Connection Machine and the *MasPar* respectively. This makes them useful as they represent language models of two of the key massively parallel SIMD computers. So as well as knowing what kind of operations to expect from a data parallel language we also are aware of which are more likely to be architecture dependent.

In this section we will briefly compare these three languages. In particular we will outline the important differences between *Lisp and TUPLE and how these can be attributed to differences in the architectures of the machines they were developed for. We will also discuss why this is not so true of Plural EuLisp and how it shares aspects with both languages.

2.2.1 *Lisp

The Connection Machine is a SIMD processor array connected to a host computer in much the same way as a conventional memory. The contents of the array can be read a word at a time by the host and the memory of the host can be accessed by the processing elements of the Connection Machine. In effect the Connection Machine and its host form a single unified address space. The operation of the Connection Machine is controlled by the host which executes programs containing both serial and parallel instructions. Serial instructions manipulate data in the memory of the host and parallel instructions are sent to the micro-controller which broadcasts the appropriate nano-instructions to the processing elements.

The organisation and operation mode of the Connection Machine is very evident in *Lisp. A parallel program in *Lisp is composed of ordinary lisp functions which contain calls to parallel functions. These functions allocate and manipulate objects in parallel on the processor array. The parallel variables can be of various types including `front-end`, in which case the contents of the pvar is the address of an object on the host.

With respect to inter-processor communication the primitives in *Lisp naturally correspond to the various styles and modes of communication that the Connection Machine can support.

2.2.2 TUPLE

In contrast to the Connection Machine the *MasPar* forms a self-contained sub-system. As well as broadcasting instructions to the processing elements the array control unit (ACU) is also capable of independent program execution and has a limited amount of local memory. The host supplies UNIX

services to the *MasPar*, e.g. job management. Although the *MasPar* is capable of independent execution most programs will require some front-end code for tasks like input, output and visualisation. In general then, an application will have a program running on the host which makes calls to the various functions on the *MasPar* which are visible to the host. The memory of the ACU and data parallel unit (DPU) are mutually accessible but the memory of the host is a completely separate address space and data must be explicitly copied between the host and the *MasPar*.

TUPLE has two distinct parts: a conventional Common Lisp system and an embedded data-parallel sub-system. The Common Lisp process corresponds to the host and the data-parallel sub-system to the *MasPar*. The `ppe` form which is used to invoke parallel execution corresponds to the `callRequest` function in mpl. Because the ACU is too small to run a full lisp system it must be run on the host and this is why the pronounced division occurs. Despite this, TUPLE does implement a uniform address space across the serial and parallel components of the system, but parallel operations on front-end references, except comparison with `eq`, are very slow.

The data parallel component in TUPLE appears to be a collection of small but complete lisp processes. That is to say, where as in *Lisp we have a single thread of control containing functions which operate on all elements of a pvar simultaneously, in TUPLE it seems the expression is executed on each processor. *Lisp is a *processor of arrays* where as TUPLE is an *array of processors* (see Section 2.3.2). This can largely be accounted for by two other important differences between the *MasPar* and the CM-2. Firstly the PEs of the *MasPar* have much more local memory than those of the CM-2 so it is realistic to have a proper, garbage collected heap on each PE. Secondly the *MasPar* also supports local indirect addressing, this means that parallel instructions can be applied to data at different addresses on different PEs and this gives a greater degree of independence between the PEs.

Where the communication primitives of *Lisp give complete access to the operations the hardware supplies, the set of functions supplied by TUPLE do not give the same control. The important differences are that TUPLE only allows values to be read from remote processors, there is no write, and the processor being read from must currently be active. Most communication operations can still be implemented using only read but sometimes this requires quite complicated manipulation of the active set.

2.2.3 Plural EuLisp

Though Plural EuLisp was developed on the *MasPar* its execution model is much the same as that in *Lisp, a serial lisp process calling functions that control the processor array. However the interpretation of the parallel conditional `if-s` is different from that of `*if` so that a parallel version of

a function resembles its serial counterpart more closely. Assuming *Lisp can support list operations consider these parallel versions of `list-length`.

```
(defun list-length (list) ;Version usable by TUPLE
  (if list
    (+ 1 (list-length (cdr list)))
    0))

(defun !!list-length (lists) ;*Lisp version
  (*if lists
    (when (|= (!! t))
          (!!+ (!!list-length (!!cdr lists))))
    (!! 0)))

(defun list-length-s (list-s) ;Plural EuLisp version
  (if-s list-s
    (+-s 1 (list-length-s (cdr-s list-s)))
    0))
```

The serial thread of control in Plural EuLisp interacts implicitly with the active set while this must be done explicitly in *Lisp, also many of the functions in Plural EuLisp automatically *bang* serial arguments to parallel functions. As a result `list-length-s` can be derived from `list-length` simply by changing the function names. This gives Plural EuLisp a measure of the *micro-macro equivalence* which is discussed in Section 2.3.2.

The more obvious difference is the addition of a processor management system. The programmer can allocate a set of processors leaving the remaining processors available for future allocation. This means that a program no longer has to control the entire processor array but just a set of processors matching the problem size. In addition, because each conformant set has its own, internal context which persists between the function calls, the sets are independent and operations on different sets can be inter-leaved without danger of interference. This has made a multi-user version of Plural EuLisp possible where any EuLisp process on the local area network can connect to a Plural server on the MasPar and perform data-parallel operations. This gives better utilisation of resources for Lisp than the MasPar job-swapper can. The Connection Machine has a separate micro-controller for each 4K PE cluster and this makes it possible for each cluster to execute a different instruction stream. Plural EuLisp gives a similar, finer grained ability to partition the array between users, but without

the performance of independent program execution possible with additional hardware.

Finally Plural EuLisp has a high-level abstraction of communication which neither *Lisp or TUPLE have. The mapping mechanism is almost identical to that in the Paralation Model and can be very powerful, it does not give access to all communication patterns and the cost of `move` can be unpredictable and implementation dependent. The merits of mappings are discussed in detail in the next section (2.3.3).

2.3 A Critique of the High Level Languages

We now look at the remaining languages, Connection Machine Lisp, Paralation Lisp and NESL which all have high-level abstractions of processor allocation, parallel execution and communication. We will examine the languages under each of these topics.

2.3.1 Processor Allocation

The paralation probably represents the clearest mechanism of processor allocation. A new set of processors of a given size can be allocated using `make-paralation` and data can then be allocated on each of the processors to create fields.

In Connection Machine Lisp an element of a xappings index can be thought of as a label for a processor and the corresponding value as data allocated in the memory of that processor. So for every Lisp object that is used as part of a xapping index there is an unique processor associated with it. Whenever a new object is used as part of an index, i.e. an object not previously used, then effectively a new processor is allocated. This is a very abstract mechanism and many programmers will no doubt be unaware of this interpretation of their operations.

Although this system of labelled processors is a very smooth mechanism if well implemented [63] (see Section 6.1.1) there are some disadvantages. For example if a particular index is used frequently there will be a lot of objects associated with the processor it labels, this can lead to the array becoming unevenly loaded. In fact the implementation avoids this by spreading values evenly across the array and storing them with a pointer to the processor. The values are sent to this processor when a computation is done, so this gives better utilisation of memory at the cost of extra communication. So though we do effectively *allocate* processors when using CM-Lisp we are really always using the *entire* array as a kind of parallel hash table/associative memory. This is quite different from Paralation Lisp where the paralation can be used to create disjoint and independent subsets of the processor array.

NESL embodies the vector random access (VRAM) model [6] and as such there is no real concept of processor allocation because the vectors are considered to be primitive. Certainly there is little concept of where one vector is in relation to another. Instead all vectors are viewed as *starting at the same place.* This seems quite reasonable for a language which is aimed rather more at vector processors than massively parallel computers.

2.3.2 Computation

The method used in Connection Machine Lisp to indicate parallel execution is again very abstract and has a strong algebraic feel to it. An interesting feature of the language highlighted by Steele and Hillis [61] is that the notation gives two points of view of parallel computation. On the one hand, it can be understood as a computation with a single thread of control, operating on arrays of data. This gives a global view of how data is being transformed, as in FP [2] and APL [14]. On the other hand it can be understood as an array of processors with each processor executing the same code that follows an α. Consider again the function `en-to-one` which we used in Section 2.1.2, this time creating a list of the same length on each processor.

```
(defun en-to-one (n)                ;Serial definition of en-to-one
  (if (= n 0) ()
    (cons n (en-to-one (- n 1)))))

α(en-to-one 5)
⇒{ → (5 4 3 2 1)}
```

Strictly speaking this is a xapping whose value is a *zillion* lists but we can consider it to be a list on every processor (though a good implementation would probably not actually do this). In this case we think of each processor independently executing a copy of the function `en-to-one`. We perceive the `ppe` form of TUPLE in much the same way. Another CM-Lisp version on `en-to-one` could be:

```
(defun en-to-one (n)
   (αif (= n 0) α()
      (αcons αn (en-to-one (- n 1)))))
```

This version we view as a serial lisp function manipulating collections of data, this is the same view we have of execution in *Lisp. The CM-Lisp • operator allows us to flag data that may differ between processes. This means that code written for a single processor simply has to be annotated with α and • to operate on a processor array. So CM-Lisp supports both microscopic and macroscopic views of parallel computation. These two views are described by Bougé [11] as:

- In the *macroscopic* view, we have a sophisticated sequential processor with the ability to operate on arrays instead of scalars: *a processor of arrays.*

- In the *microscopic* view, we have an array of elementary sequential processors operating in parallel on their private scalar data. An external sequencer is in charge of synchronising them: *an array of processors*

With the aid of a simple but essentially complete SIMD language, called L, he goes onto show that the microscopic and macroscopic views are related in an intrinsic way. I use the phrase *micro-macro equivalence* to refer to this property. If a language is micro-macro equivalent then the programmer can code for a single processor and then scale the operations to as many sites as are required, it is often simpler to program *in the small* rather than manipulating active sets and using operations applied to the entire array.

Although `elwise`[2] in Paralation Lisp is not as general as α it still has this property. The `elwise` form has the appearance of a serial lambda expression, that is a code segment to be executed on a single processor. The number of processors it is actually executed on is dependent on the parallel arguments it is applied to. In the same way that • is used to flag data which is already parallel, the argument list of `elwise` represents a list of variables which are to be considered *already parallel.*

As well as having this useful property, `elwise`'s simplicity makes its operation much easier to understand. We can interpret `elwise` as, *execute this expression in parallel and bind these identifiers to their local values rather than the entire field,* so `elwise` can be thought of as a kind of parallel `let` statement. Since `let` is simply syntactic sugar for a lambda closure, and borrowing from Common Lisp we could rewrite an `elwise` expression as follows:

```
(elwise ((a A) (b B)) (+ a b)) ≡ (pfuncall (lambda (a b) (+ a b)) A B)
```

Where `pfuncall` behaves like `funcall` but invokes parallel execution. But given that fields are sequence type objects we could use a more familiar function whose meaning would be more accessible to most lisp programmers:

```
(map-field (lambda (a b) (+ a b)) A B)
```

This is not strictly correct because `elwise` also permits updates of the fields it applies the expression to, something most mapping functions do not support. However there are clearly similarities between a parallel call with a sequence data structure and mapping a function over the data structure.

[2] `Elwise` is essentially the same as `over` and so will not be explicitly considered here.

Because an `elwise` expression *declares* a set of variables as *already parallel* for the entire expression it is rather simpler to use than CM-Lisp.

To determine if something is parallel or singular in Paralation Lisp one need only look back to the enclosing `elwise` form. Where as in CM-Lisp one may have to keep track of perhaps several levels of $\bullet$'s and α's to determine what state an object is in. This is further complicated by it being possible for the programmer to introduce parallelism at any level as long as it is balanced correctly throughout the expression. As a result two apparently different annotations can be equivalent programs though this may not be at all obvious.

Another drawback with CM-Lisp is its automatic selection of context, although `xunion`, `over` etc. make it possible to perform operations on subsets of a collection, it can often be verbose. In contrast, it is fairly easy to ignore unnecessary sites in a paralation and these can be eliminated when a final result is produced.

2.3.3 Communication

First we will consider the reduction and scan functions that the languages supply. Implemented correctly the complexity of these functions is $O(log_2 n)$ and Guy Blelloch argues the complexity can be considered $O(1)$ on certain architectures. This makes them a very important part of data parallel programming. The reduction operators of Paralation Lisp and CM-Lisp are very similar, both `vref` and the unary form of β can use any binary combining function to reduce a field or xapping. NESL does not have a general reduction operator, instead it supplies a separate function for each of `+`, `max`, `min`, `or` and `and`. NESL also supplies scan functions for each of these operators. There is no specific scan mechanism in CM-Lisp or Paralation Lisp and they must be implemented using the general communication mechanisms.

Mappings and the binary form of β perform similar kinds of communication, though not identical. In particular β can only support *many to one* patterns whereas `match` can also create *one to many* mappings. Although mappings can support more patterns than β, `match` cannot create every possible communication pattern between paralations. For example the pattern required for a prefix operation cannot be created by `match`. However if `match` were to accept the predicate for comparing the source and destination objects, then this map could be created by giving `<` as the predicate and matching the index fields. Figure 2-3 illustrates the patterns that `match` and β can, and cannot support.

Although we think of a mapping as *a bundle of arrows* between paralations it is, in fact, a relation between fields, as a result we cannot create all possible patterns of arrows using `match`. Perhaps a more important difference is that the communication patterns defined by `match` are reusable and any

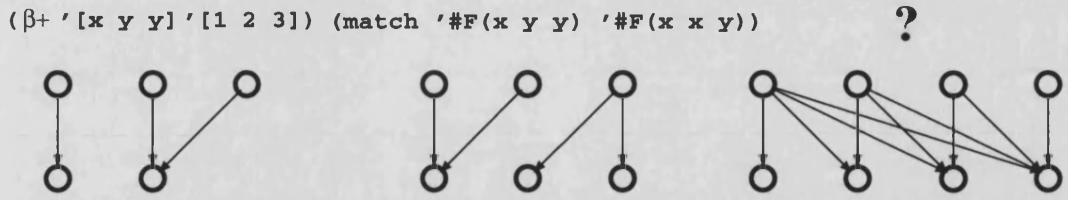


Figure 2-3: Communication Patterns Supported by β and Mappings

work associated with the creation of a mapping need only be done once. In fact the implementation of CM-Lisp cleverly minimises the work associated with β (though there are still some problems, see section 6.1.1), but this is still a significant advantage for Paralation Lisp.

NESL does not have a high level abstraction of communication and simply provides a library of permutation functions like `permute`, `get`, `put` etc. Placing this restriction on the patterns of communication that can be achieved with a single operation allows for a very efficient implementation but does not constitute a real abstraction of inter-processor communication.

2.3.4 Summary

Though NESL is a high-level language it does not abstract processor allocation as we would like it to because it considers vectors to be primitive and they do not have any real sense of location. Neither does it have any real abstraction of communication but simply provides functions for various useful types of permutation. As `over` is almost identical to `elwise` it cannot really be considered a contribution to language design either.

At the other end of the scale we have the extremely abstract Connection Machine Lisp. Though the language captures many important aspects of data parallelism it often seems *too* abstract. Determining what is already parallel and what has been made parallel can mean keeping track of many levels of $\bullet$'s and α's which can be inserted seemingly anywhere in the code. The implicit selection of the intersection can also lead to extra manipulations and the language has to supply additional functions like `xunion` and `over` to make this possible, which denies the completeness of the three kernel operators. Lastly, only being able to access communication by the mechanism of creating a new xapping seems insufficiently expressive for computers with the capabilities of the Connection Machine.

Paralation Lisp seems to lie between these two extremes. It has a simple, clear and precise mechanism for allocating sets of processors on which code can then be executed in parallel. `Elwise` itself is a simple and familiar mechanism and it is clear what parts of an expression change between processors and which are invariant. Inter-processor communication is abstracted by mappings which

are effectively defined by specifying which sites should be connected to each other.

Thus, in a limited way at least, Paralation Lisp meets our requirements of being able to allocate and connect processors as we need to, something which is lacking in both NESL and Connection Machine Lisp.

2.4 Meeting the Requirements

We have now looked at the functional languages suited to massively parallel SIMD architectures like the Connection Machine and found Paralation Lisp best meets the requirements of active memory programming we identified in Section 1.2.2, namely:

> *We can manipulate processors and communication links with the same ease we manipulate memory and pointers.*

This should mean we can allocate processors as we need them and are able to define and reconfigure the communication links between them in software. Paralation Lisp has a clear concept of processor allocation and mappings give us a simple way of connecting processors to each other. But is this really what we expect of an active memory language?

Only in part: we can definitely use paralations and mappings to represent data structures with processors and communication links which can the be operated on in parallel. But this is rather different from constructing active data structures which is what the requirement suggests. An example of this was encountered when building a data parallel implementation of a connectionist network [15] (this is covered in more detail in Section 4.4).

Briefly the connectionist networks [22][3] experimented with consisted of a graph with weighted arcs constructed from a collection of declarations of related objects, for example:

(**Object** Gimli **is** Dwarf
    nature good
    is_fond_of fighting)

Further definitions would be made for entities like good, fighting etc. Each object is represented by a node of the graph and the arcs correspond to the properties of the objects. This network can then be used to make deductions by weighting nodes of interest and *running* the network. To do this each node calculates a new value based on the weights of the nodes and arcs it is connected to. This process is repeated for a fixed number of iterations or until the network stabilises. On completion the network can be interrogated to find any relationships the network has identified.

[3] Note this is a rather specific example of connectionist networks.

In order to build a representation of the network on the *MasPar* the network had to be built first in the host memory. A paralation of the correct size could be then allocated and a set of mappings created which matched the connectivity of the network. But surely we could have built the network in the active memory directly rather than having to use an intermediate representation. This highlights a general problem with the Paralation Model, it is difficult to actually *construct* active data structures using paralations rather than simply *representing* them. A simpler example is the function `field-append-2` which takes two fields and concatenates them:

```
(field-append '#F(a b c d e) '#F(f g h i j))
⇒ '#F(a b c d e f g h i j)
```

Now consider how we would implement this function in Paralation Lisp, this EuLisp version is based loosely on code given by Sabot:

```
(defun field-append-2 (fld-1 fld-2)
  (let* ((size-1 (length fld1))
         (total-size (+ size-1 (length fld2)))      ;New Paralation big
         (new (make-paralation total-size))         ;enough for both fields
         (to-front (match new (index fld-1)))       ;fld-1 mapped to front
         (to-back (match (elwise (new)              ;fld-2 mapped to
                           (- new size-1)))))       ;just after fld-1
    (elwise ((front (move fld-1 to-front () 'void))
             (back (move fld-2 to-back () 'void)))  ;Move the fields
      (if (eq front 'void) back front))))           ;and merge results
```

What seems to be a relatively simple task requires two `match` and two `move` operations, and we have to allocate a new paralation to contain the result plus two intermediate fields. This is because paralations cannot be constructed from existing paralations, a completely *new* set of sites must be allocated and then the data must be moved into this new paralation. The Paralation model's solution to this is to supply a library of *useful* functions for manipulating paralations: all these functions can be implemented in Paralation Lisp but they can be fairly expensive. Making them part of a standard library means they can be implemented more efficiently at a lower-level than Paralation Lisp. This is a working solution but it diminishes the completeness of the Paralation Model's small and simple kernel.

This is similar to the situation in CM-Lisp where the need for other functions like `xunion` diminishes the completeness of the three operators α, β and $\bullet$.

2.5 More About Paralation Lisp

We have now established that as well as abstracting the key ideas in data parallelism, Paralation Lisp is also the nearest to fulfilling the requirements of an active memory language. Although it does not completely meet our expectations it seems a good platform for developing an active memory language, which we do in the next chapter. In this section we give some more details on the language which were not covered in the original description.

2.5.1 Value Reference

Because reduction is such an important mechanism `vref` was included in the earlier description of Paralation Lisp. However it is in fact another library function, the Paralation Lisp definition of `vref` given below is based loosely on that given by Gary Sabot:

```
(defun vref (fld with . else)
  (if (zerop (length fld)) else           ;Handle empty fields
    (field-ref
     (move fld (match (make-paralation 1) ;Move all the values to
                      (elwise (fld) 0))   ;a single location and
           with ()) 0)))                  ;return its contents
```

2.5.2 Expand

`(expand` *field(field))* → *field*

The function `expand` concatenates the elements of *field* into a single field creating a new paralation of the appropriate size in the process. The order of the sub-field values in the result is based on the index ordering of *field.*

```
(expand '#F(#F(A B) #F(C) #F() #F(3 2 9 0)))
⇒ #F(A B C 3 2 9 0)


(defun expand (field)
  (vref field field-append-2 (make-paralation 0)))
```

2.5.3 Choose

`(choose` *field(bool))* → *mapping*

`Choose` creates a new paralation with an element for each non-`nil` value in *field(bool)*. It then creates a mapping connecting the non-`nil` sites to their counterparts in this new paralation.

```
(setq p (make-paralation 5))
⇒#F(0 1 2 3 4)
(setq select (choose (elwise (p) (odd p))))
⇒#<mapping>
(move p select () ())
⇒#F(1 3)
(move (elwise (p) (list-ref '(a b c d e) p))
      select () ())
⇒#F(b d)
```

Below is an implementation of `choose` in Paralation Lisp which uses `expand`. This works by converting the field to a field of fields. Each sub-field is a singleton paralation for every non-`nil` element and an empty paralation otherwise. Concatenating these sub-fields using `expand` gives a paralation of the correct size which is then matched to the boolean field to give the desired mapping.

```
(defun choose (field)
  (let ((point-back
         (expand
           (let ((i (index field)))
             (elwise (field i)
               (if field
                   (elwise ((p (make-paralation 1))) i)
                   (make-paralation 0)))))))
    (match point-back (index field))))
```

2.5.4 Collapse

`(collapse` *field*`)` → *mapping*

This function is similar to `choose` but the destination paralation contains an element for each distinct element in the argument field. The mapping connects the sites in the source to their counterpart in this new paralation.

```
(setq name '#F(a a d a b))
⇒ #F(a a d a b)
```

```
(setq map (collapse '#F(a a d a b)))
⇒ #<mapping>
n(move (elwise (name) 1) map + ())
⇒ #F(3 1 1)
```

In the implementation below a representative (the first) is chosen for each distinct value in the field. Matching the field to itelf and moving the index down the resulting map with `min` gives each site the position of this member of its sets. Comparing this position to the objects index identifies these *special* elements. `Choose` creates a paralation of the correct size with a location for each distinct value. All that remains is to create the mapping.

```
(defun collapse (fld)
  (let* ((i (index fld))
         (min-holder (move i (match fld fld) min ()))
         (min-holder-p (elwise (min-holder i) (= min-holder i)))
         (distinct-vals (move fld (choose min-holder-p) () ())))
    (match distinct-vals field))
```

2.5.5 Collect

`(collect` *field mapping*`)` → *field*

This is similar to `move` but colliding values are collected into sub-fields, so no combining function is needed. If no values arrive at a site then the result is an empty field, so a default value isn't needed either. So using the mapping created using `collapse` previously:

```
(collect (index name) map)
⇒ #F(#F(0 1 3) #F(2) #F(4))
```

The implementation below works by first turning the input field into a field of singleton fields. Moving this field down the mapping with `field-append-2` as the combining function and the empty field as a default value has the desired effect.

```
(defun collect (field map)
  (let ((to-key (mapping-to-key)))        ;Extract destination from mapping
    (move (elwise (field)                 ;so as to create default field
            (elwise ((p (make-paralation 1))) field))
          map field-append-2
          (elwise (to-key) (make-paralation 0)))))
```

This operation is similar to the `partition` operation of NESL with `expand` having the same rôle as `flatten`.

2.5.6 Fields as Sequences

Many languages, for example Common Lisp and EULISP, have generic function mechanisms and in particular they may have sets of generic functions that are applicable to any general sequence data type, i.e. lists, vectors etc. As fields are a type of sequence these functions can be extended to operate on fields as well. This has been done in the Common Lisp based implementation and where possible the functions have been implemented in parallel. These functions are not necessary and are supplied simply for convenience. Some examples are:

`(elt` *sequence index*`)` → *obj*

`(subseq` *sequence start end*`)` → *sequence*

`(reverse` *sequence*`)` → *sequence*

`(make-sequence` *type size*`)` → *sequence*

These functions need little explanation, but very briefly: The function `elt`'s behaviour is the same as that of `field-ref`. The function `subseq` extracts a subsequence, when applied to a field this will in general require a new paralation to be created. The contents of a field can be `reversed` by using a mapping from the field's paralation to itself. And finally the function `make-sequence` is essentially the same as `make-paralation`.

Similarly, the implementation also extends the generic string and set functions, allowing fields to be treated as sets and strings. These functions do not enhance the language in any way and have been mentioned here purely for completeness.

Chapter 3

Extending Paralation Lisp

In the previous chapter we looked at some of the functional languages for massively parallel architectures like the Connection Machine. Of those we considered, the Paralation model has simple, clear abstractions of processor allocation, parallel computation and inter-processor communication but it does not really fulfil the requirements of an active memory language. We can define active memory style operations using Paralation Lisp but these are often verbose. This is remedied by a library of *useful* functions like `field-append-2` and `choose` which can be efficiently implemented at a level below Paralation Lisp. Though these functions can all be implemented in Paralation Lisp they are cumbersome and this suggests the kernel of the paralation model is not sufficient for the needs of active memory programming. In this chapter we look at some existing extensions to Paralation Lisp that address some of the deficiencies in the model. We finish by presenting a new set of *active memory* extensions to Paralation Lisp.

One feature of the Paralation Model which makes it attractive as a basis for further development is its abstraction of locality, something poorly represented in the other languages. The locality of processing sites defines how easily they can communicate with each other, i.e. the cost for two processors to communicate will be less if they are close to each other. This is analogous to what is defined by an active data structure, if we connect two processors with a communication link we are indicating that they should be able to communicate easily. This suggests that locality issues may be worth exploring further as a direction for realising an active memory language.

Paralation Lisp, as it stands, has a very coarse concept of locality, sites are either *near* to each other if they are in the same paralation or *far* apart if they are in different paralations. Although this is a good start we would prefer a finer grain model of locality, one that operates at the site level rather than the paralation level.

3.1 Shaped Paralations

Shaped paralations are an extension to Paralation Lisp defined by Gary Sabot [55, Ch. 5]. It is useful for a Paralation to have shape if it is being used to model a problem where the locality of the data is significant. There are various situations where this is the case, for example many vision algorithms are based on a grid of data performing computations on *neighbourhoods* of grid cells. If the compiler is aware of the paralation's shape, it can map the sites onto the processors so that the physical arrangement of the sites matches their logical arrangement.

Giving paralations shape can also make them easier to work with. Communication operations can be defined which reflect the shape of the paralation, for example we may wish to shift the values of a field in a grid-shaped paralation one position north. How the elements of a paralation are accessed can also be based on its shape, we may wish to specify an element of a grid-shaped paralation by its (x, y) coordinate rather than its index position.

Thus the shape mechanism in Paralation Lisp has two components, one defining locality and the other defining access. We will now give a brief description of this shape facility, a full description can be found in [54].

3.1.1 Shape Locality

To define the locality of a shape the user specifies what kind of communication within the paralation should be inexpensive. Essentially this specifies which sites of the paralation are *near* to each other. So in a grid shaped paralation we would probably expect each site to be near its west neighbour and that shifting a field west should be an inexpensive[1] operation.

To do this a paralation is allocated and mappings corresponding to the inexpensive communication operations are created for that paralation. A new paralation can then be created where the physical sites have been arranged so that these mappings will, hopefully, be more (but never less) efficient. Calling the function `make-shaped-paralation` with a list of the locality defining mappings creates the new paralation and returns its index field. Below we create a rectangular paralation with 4 columns and 4 rows where each element, except those at the edges, has four immediate neighbours.

```
(setq width 4)              ;To clarify code
(setq rank 4)
(setq p (make-paralation (* width rank)))
(setq col (elwise (p) (+ (remainder p width) 1)))
```

[1] Whether it is efficient will depend on the architecture and implementation.

```
(setq rectangle (make-shaped-paralation
                 (list (match (elwise (p) (- p width)) p)
                       (match (elwise (p) (+ p width)) p)
                       (match (elwise (p col) (if (= col 1) () (- p 1))) p)
                       (match (elwise (p col)
```

When the new paralation is created each of the locality mappings are automatically created for the new paralation. The new mappings are associated with the new paralation and accessed using the function `shape-map`:

`(shape-map` *field subscript*)

Where *field* belongs to the new paralation and *subscript* is the position of the original mapping in the list passed to `make-shaped-paralation`. It is not strictly necessary for the mappings to be created in this way as the user can simply recreate them and they should automatically be faster on account of the better paralation allocation. However it is time saving and also allows the implementation to return special mappings taking advantage of the underlying architecture in a way that the general mappings created by `match` may not be able to. We can now define functions to perform grid-based shift operations on fields in the same paralation as `rectangle`:

```
(defun N (f edge) (move f (shape-map f 0) () edge))
(defun S (f edge) (move f (shape-map f 1) () edge))
(defun E (f edge) (move f (shape-map f 3) () edge))
(defun W (f edge) (move f (shape-map f 2) () edge))
```

Hopefully the paralation `rectangle` will have been allocated to take advantage of the underlying architecture – on a processor array we would expect the elements to be arranged in a grid and the mappings `N`, `S`, `E` and `W` to be using the nearest neighbour communication network. We can now define a function to find the average of each site's four neighbours making use of the paralation's shape.

```
(defun average (value)
  (elwise ((north (N value 0.0))
           (south (S value 0.0))
           (east (E value 0.0))
           (west (W value 0.0)))
    (/ (+ north south west east) 4.0)))
```

3.1.2 Shape Access

Defining the internal locality of a paralation certainly gives it some kind of *shape* but this is only obvious when moving data around within it. To complete the appearance we can also modify how it is printed and how its elements are accessed.

The shape access is controlled by the function `define-shape-access` which is used to associate various pieces of information with the paralation. The syntax is:

`(define-shape-access` *field init-option**`)`

Where an *init-option* is a symbol followed by a corresponding value. The possible symbols and their values are:

`shape-info:` This allows any appropriate data to be associated with the paralation. It can be accessed by applying the function `shape-info` to any field in the paralation. Here `shape-info` is used to associate the shape type and dimensions with the paralation.

```
(define-shape-access rectangle 'shape-info '(grid ,width ,rank))
```

`site-names:` This allows a special field to be associated with the paralation which can be obtained by applying the function `site-names` to any field in the paralation. The idea is that this field can be used as an alternative index field for the paralation, one that reflects its shape. For example, we may wish to identify each site of `rectangle`'s paralation by its (x, y) coordinate.

```
(define-shape-access rectangle 'site-names
                     (elwise ((i rectangle))
                       (list (/ i width) (remainder i width))))
⇒ #F((0 0) (0 1) (0 2) (0 3)
     (1 0) (1 1) (1 2) (1 3)
     (2 0) (2 1) (2 2) (2 3)
     (3 0) (3 1) (3 2) (3 3))
```

`print-function:` This specifies a function to print fields that belong to the paralation. This allows us to define an output format that reflects the shape of the paralation. In the previous code segment the elements of the `site-name` field have been arranged in a rectangle, this would be done by an appropriate `print-function`.

`field-ref:` This allows us to change the way the elements of a paralation are referenced. For our grid paralation we would like to specify the (x, y) position of the element rather than its index.

Below we define a function which will access the elements of the paralation by comparing the given (x, y) coordinates to the paralation's `site-names`.

```
(define-shape-access rectangle 'field-ref
                     (lambda (f x y)
                       (vref (elwise ((x-y (site-names f)) f)
                               (if (and (= x (first x-y))
                                        (= y (second x-y))) f ()))
                             (lambda (a b) (if a a b)))))
```

If a special accessor has been defined for a paralation then `field-ref` passes its arguments to that function, otherwise it simply behaves in its default fashion. This version of `field-ref` uses a reduction to select the correct value. A more efficient method would be to work out the index position and then use the default version of `field-ref` to access it:

```
(define-shape-access rectangle 'field-ref
                     (lambda (f x y)
                       (default-field-ref
                        f (+ (* (second (shape-info f)) y) x))))
```

Shaped paralations are a useful programming paradigm greatly simplifying the programmer's task of matching processing sites to the problem. It also allows the implementer to make the facilities of the underlying architecture available transparently to the programmer at the Lisp level. Any Paralation Lisp implementation should include a standard library of shapes and where appropriate these can be implemented to take full advantage of a particular platform.

The shaped paralation mechanism can be interpreted as allocating collections of processors and gluing them together with communication links and so at a superficial level meets the requirements of our active memory language. But it is somewhat limited, rather than allocating sites as they are needed and connecting them with communication links we allocate an entire collection of processors and then define communication patterns which match the desired structure. We can make an analogy between paralations and arrays in a language such as C, we could enhance the use of arrays by giving them shape, but this would not give them the same utility as dynamic C data structures. It also suffers from the need to create new paralations: if we wish to add an additional site to the structure we must reallocate the paralation and redefine all the mappings. Even simply changing the connectivity will require rebuilding some of the mappings, and this may be non-trivial.

Shaped paralations are not proposed as a constructive paradigm so it is inappropriate to comment on their effectiveness in this area. But consider the case where we have two rectangular paralations with edges of equal length. It seems quite reasonable that a programmer would want to join these into a single rectangular paralation. However this is not a simple process, we must create a new paralation of sufficient size and `move` the original paralations into it as we do in the field append function. But now the locality and access information will have been lost and must be regenerated.

In the next section we look at a system which addresses the problem of shaped paralations containing shaped paralations, but as a decomposition operation rather than a constructive one.

3.2 Paralation Views

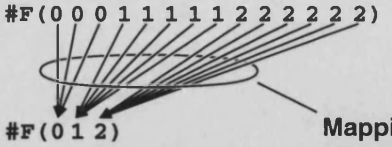
Paralation Views are an extension to the Paralation Model devised by K. Goldman [24]. They enhance shaped paralations by allowing a paralation to be viewed as multiple different shapes. A *view* is a partition of the sites of a paralation into a set of *classes*. Each of these classes is itself a paralation and the view is represented as a nested paralation with one element for each class.

A similar kind of partitioning of a paralation can be achieved using the Paralation Lisp library functions `collect` and `collapse` (see Sections 2.5.5 and 2.5.4). The function `collapse` accepts a field and creates a new paralation with a site for each distinct object in that field. The result is a mapping from the original field to the new paralation. Below we collapse a paralation of 15 elements into one with three elements:

```
(setq set (make-paralation 15))
⇒ #F(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14)

(setq set-ids (elwise ((i set))
                (list-ref '(0 0 0 1 1 1 1 1 2 2 2 2 2 2 2) i)))
⇒ #F(0 0 0 1 1 1 1 1 2 2 2 2 2 2 2))

(setq map (collapse set-ids))
⇒ #<mapping>
```



The function `collect` is a `move`-like operation where all collisions are combined by collecting them into a new field, so the result is a field of fields and this is effectively a partition of the original paralation:

```
(collect '#F(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14) map)
⇒ #F(#F(0 1 2) #F(3 4 5 6 7) #F(8 9 10 11 12 13 14))
```

However, strictly speaking, this is not a partition of the original paralation, as this would suggest the sites of the paralation had been split into subsets. All the new paralations are composed of completely new sites which form sets equivalent to a partitioning of the original paralation. Figure 3-1 illustrates the difference between this and a true partitioning of the paralation. In the Paralation Model new paralations with 3, 5 and 7 elements are allocated plus an additional paralation of 3 elements to hold each of these paralations. With Paralation Views a 3 element paralation is allocated to hold the result, but the paralations representing the classes of the partition are composed of sites in the original paralation.

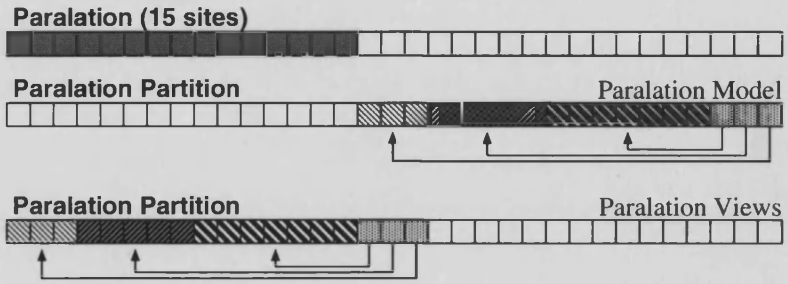


Figure 3-1: Different mechanisms for partitioning paralations

As discussed in Section 2.5.5 moving fields from the parent paralation into the partition created using `collapse` and `collect` requires communication and a new set of mappings. But for the partition created using a view no communication is required because the appropriate value is already on the processor associated with each site in a class paralation.

The partitions created by `collapse` and `collect` also have another drawback. If we have two fields of values which we need to use on a partition of the paralation, simply `collecting` both fields with the appropriate `collapse` mapping, will actually place them in different, new paralations. So we cannot use `collapse` and `collect` to project several fields into a partition. Instead the partition must be created once, and a mapping then created which will project fields into the partition.

The ability to partition a paralation is very useful, for example a grid can be viewed as a collection of rows, allowing an individual row to be operated on. Paralation Views reduce the cost of using such partitions by removing the need for new sites, and hence the communication cost of moving data into the partitioned paralation. In addition to this it is possible to have multiple views on the same paralation, so for example a grid paralation could be viewed both as a collection of rows and as a collection of columns.

3.2.1 Creating Views

There are three different ways of creating a view of a paralation and more than one view of a paralation can exist at a time. The most general method is `extract` which permits completely arbitrary subsets to be defined. The functions `split` and `project` specify partitions based on the coordinate systems of `grid`-shaped paralations. The Cartesian grids are part of the shape library and so this may have been implemented to take advantage of the underlying architecture. Because the `project` and `split` partitions are based on the coordinate system the class paralations will also benefit from any improved arrangement of the physical sites.

To illustrate the three methods we will use a 4×4 `grid`-shaped paralation created using the Paralation Lisp shape library function `make-grid`. This creates grid-shaped paralations of any dimension, the lengths of each dimension are given in a list (note that this is slightly different from the Common Lisp code given in [54]).

```
(setq mat (make-grid '(4 4)))
⇒ #F((0 0) (0 1) (0 2) (0 3)
     (1 0) (1 1) (1 2) (1 3)
     (2 0) (2 1) (2 2) (2 3)
     (3 0) (3 1) (3 2) (3 3))
```

The grid-based functions, `split` and `project` can be used on paralations of any dimension, but this 2-dimensional example is simpler to illustrate.

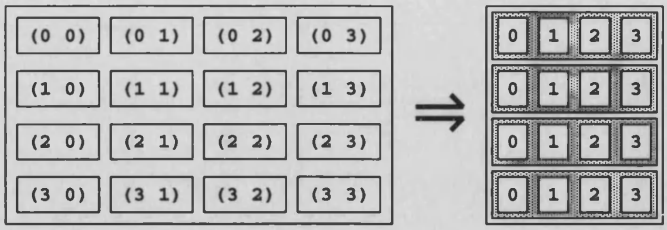
Project

A shaped paralation which has been given a coordinate system can be decomposed into a collection of paralations by *projecting* on a coordinate (or set of coordinates). The classes of the view contain elements whose value(s) for that coordinate (or set of coordinates) are equal.

Figure 3-2 shows a view created by projecting on the first coordinate of the 4×4 grid-shaped paralation `mat`. The coordinates to project on are specified by a list of booleans which match the format of the site names, here the site names are of the form $(x_1\ x_2)$, so the list `(t nil)` specifies projection on the first coordinate. The third argument specifies a shape for the classes of the view, this associates a set of (efficient) predefined mappings with each class paralation. In this case, the `ring` shape connects each element to its two immediate neighbours, wrapping round from the last element to the first.

The resulting view has a class for each of $x_1 = 0$, $x_1 = 1$, etc. so it has 1 dimension. In general a view has a dimension for each projected coordinate, but in this case only one coordinate was projected

on. To access the classes of a view a version of `field-ref` matching the views dimension must be used, in this case a simple `field-ref` is appropriate.




```
(setq rows (project mat '(t nil) 'ring))
```

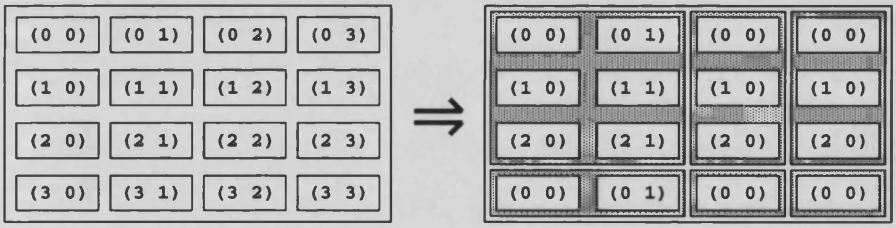
Figure 3-2: Creating a Paralation View using `project`

Projection is useful for isolating individual planes of multi-dimensional structures to be operated on in parallel without the need to move the data into a new paralation.

Split

A view can also be created by specifying a partition of each coordinate axis in a shaped paralation. To do this a list is given for each coordinate axis which specifies where to make the *cuts*.

This divides each axis into a set of sections, each axis which is cut in this way forms an axis of the resulting view. In Figure 3-3 the first axis is divided into three and the second axis into two, so the shape of the resulting view is a 3×2 grid. As with `project` an appropriate version of `field-ref` must be used to access the classes of the view, in this case a 2-dimensional version is needed.




```
(setq sub-mats (split mat '(2 3) '(3) 'rectangle))
```

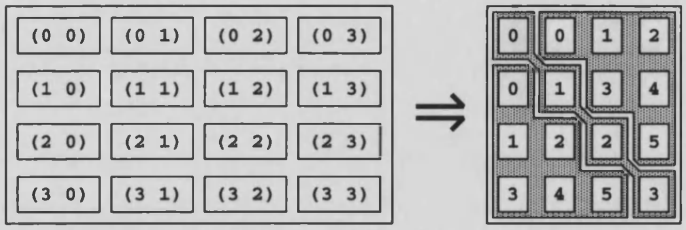
Figure 3-3: Creating a Paralation View using `split`

`Split` is useful for divide and conquer algorithms, for example image processing algorithms operate by repeatedly dividing an image into halves. In such a case each class would also need to be a rectangle and `split` allows the shape to be specified in the same way as `project`.

Extract

Extract is the most general way of creating a view. The partition is specified by a *decider* field of non-negative integers: each class is made up of the sites with equal decider values. So the resulting view has a class for each distinct value in the decider field. This is very similar to the operation performed using `collapse` and `collect` in Section 2.5.5.

In figure 3-4 we use `extract` to decompose `mat` into its diagonal, upper-diagonal and lower-diagonal components. The decider field is created by comparing the x and y coordinates for each site:




```
(extract (elwise ((x-y (site-names mat)))
           (cond ((= (first x-y) (second x-y)) 0)
                 ((> (first x-y) (second x-y)) 1)
                 (t 2)))
         'unshaped)
```

Figure 3-4: Creating view using `extract`

The values in the decider field are important as these correspond to index positions of the classes in the paralation representing the view. Hence in this example element `0` is the diagonal, element 1 the upper diagonal and 2 the lower diagonal. This is entirely arbitrary and any other ordering could have been specified by using different numbers in the decider field. If the decider values are not contiguous then empty classes are created for the values missing from the decider field. As a result the size of an `extract` view is one greater than the maximum value in the decider field.

3.2.2 Operating on Views

Each class paralation shares fields with its parent paralation. To operate on the elements of a parent field within a class paralation the function `take` is used to obtain the portion of a parent field that belongs to a class:

(`take` *parent-field class-field*)

For example, below we use the view `rows` to extract the fourth row of the field `mat`, this could of course be any field in the same paralation as `mat`.

```
(take mat (field-ref rows 3))
⇒ #F((3 0) (3 1) (3 2) (3 3))
```

Paralation Views extend the syntax of `elwise` to allow `take` to be specified in the field list, this is a useful shorthand which avoids creating large numbers of temporary fields.

```
(elwise (take mat (field-ref rows 3))
  (* (first mat) (second mat)))
⇒ #F(0 3 6 9)
```

Paralation views are a useful extension to Paralation Lisp making it simple to decompose a collection of objects into sub-sets which can be operated on fully in parallel. Also the sub-sets have their own coordinate system making many operations simpler and the language more modular as a whole.

Views are of interest to us because they draw attention to the importance of the actual processing sites which make up a paralation. A paralation gives the programmer a handle on a *collection* of processing sites but this abstraction is often too coarse. An operation on a single row of a matrix may be expressed best in a paralation containing only those elements, but this does not necessarily dictate the need for a new set of processing sites, just a different handle on some of the existing set.

The importance of the individual processing sites is also apparent in our requirements for an active memory programming language. Suppose we have two collections of processors which we are treating as sequences and we have a set of data allocated on each. If we wish to append these sequences we will naturally create a new, larger collection of processors, there is no obvious reason why we should want these to be new processors though. But in paralation lisp we must allocate a new paralation and perform two match and move operations. If however we were able to simply create a paralation whose sites were the union of those in the existing paralations, there would be no need for any communication and this would greatly reduce the complexity of the operation.

Views address this issue but approach it from the opposite direction. If we want to take a sub-sequence of a sequence paralation with a view, we can simply create a new handle on the appropriate set of sites, which is much cheaper then allocating a new paralation and then having to match and move the sub-sequence data into it. So although we cannot take the sites of several paralations and collect them into a single paralation, a view does allow us to take a subset of a paralation's sites and use it to make a paralation.

3.3 Elementwise Shape

In the previous two sections we saw how shaped paralations add extra locality properties and hence structure to a collection of processors. We also looked at a useful set of extensions for decomposing collections of processors into subsets. The ability to decompose a paralation drew attention to the importance of the sites in a paralation and not just the paralation in its entirety. We now look at another type of shaped paralation [43] which places more emphasis on the elements of the paralation than the paralation as a whole.

In this model the shape of the paralation is defined by giving each site of the paralation a set of *neighbours*. This is done by associating a *structure* field with the paralation in the same way each paralation has an index field. In a shaped paralation each element of the structure field is an instance of some class. Each slot of the class instance specifies the neighbouring site in that direction. So to define a shaped paralation the programmer must give a class definition and a paralation initialisation function. Thus we might define a class for a rectangular paralation where each element has four neighbours as follows:

```
(defclass rectangle ()
  ((N initarg N
      accessor N)
   (S initarg S
      accessor S)
   (E initarg E
      accessor E)
   (W initarg W
      accessor W))
  constructor get-rectangle)
```

When we allocate a paralation with a rectangular shape these slots must be initialised appropriately. Ideally we could extend the `make-paralation` syntax to allow the specification of the class and initialiser but for simplicity we define a specific allocation function:

```
(defun make-rectangle (width height)
  (let ((new (make-paralation (* width height))))
    ((setter shape) new
     (elwise (new)
       (let ((row (+ (/ new width) 1))
             (col (+ (remainder new width) 1)))
```

```
        (get-rectangle
          'N (if (= row 1) nil (- new width))
          'S (if (= row height) (+ new width))
          'W (if (= col 1) nil (- new 1))
          'E (if (= col width) nil (+ new 1))))))
    ((setter attributes) new (cons width height))
    new))
```

To create a rectangular paralation we allocate an ordinary paralation of the correct size and then create the structure field with an `elwise` expression. This field is held in an extra `shape` slot associated with the paralation object. There is an additional attributes slot which is used to store any other useful information, in this case the dimensions of the rectangle. This can be used by the field printer to display the field with the proper layout. We can now create a rectangular paralation:

```
(setq box (make-rectangle 4 3))
⇒ #F(0  1  2  3
     4  5  6  7
     8  9  10 11)
```

Two special functions, `get` and `put` are supplied to move data around within a paralation between neighbours. `Get` takes a shaped field, an accessor and a default value.

```
(get N box 'edge)
⇒ #F(edge edge edge edge
     0    1    2    3
     4    5    6    7)
(get W box 'edge)
⇒ #F(edge  0  1  2
     edge  4  5  6
     edge  8  9 10)
```

Informally we can describe `get` as *each element takes its value from the neighbour in the given direction.* The reverse operation `put` can be thought of as *each element writes its value to the neighbour in the given direction.*

```
(put S box cons 'edge)
⇒ #F(edge edge edge edge
     0    1    2    3
     4    5    6    7)
```

Notice that put takes a combining function to resolve collisions in the same way that move does. As each element only has one neighbour in a given direction get does not need a combinator, but elements can share neighbours and so put can cause collisions. Because get is a collision free operation it is potentially more efficient than move.

This version of shape also gives the implementor a way to take advantage of the underlying architecture by supplying a library of pre-defined shapes. For example on a processor array the function make-rectangle could ensure the elements of the paralation were arranged in a grid. Then the accessors in the class definition could be replaced by functions using the nearest neighbour network which would be used by get and put rather than their default behaviour, which would be to use the global router.

3.3.1 Constructing Paralations

In section 3.1 we considered the difficulties of joining shaped paralations together. Because elementwise shaped paralations hold their shape description in fields it makes it simple to create composite paralations where the local connectivity is preserved. There is still some work involved, we must create a new paralation and move the fields into it, but creating the shape of the composite field simply needs us to move the paralation shape fields into the new paralation and make this field its shape. Within the shape field the slots must be modified to reflect their position inside a larger paralation, but this simply requires adding a constant for each sub-paralation. The function join takes a set of fields in shaped paralations and creates a new paralation containing all the paralations, in the same order with their local shape preserved.

```
(setq xob (make-rectangle 4 3))
⇒ #F(0  1  2
     3  4  5
     6  7  8
     9 10 11)
(setq both (join box xob)
⇒ #F(0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 8 9 10 11)
```

The resulting paralation has no real shape of its own, it is simply a *bag* of shaped paralations, so it is not printed in any special format. However we can still use get and put to move data within the paralation.

```
(get N both 'edge)
⇒ #F(edge edge edge edge 0 1 2 3 4 5 6 7 edge edge edge 0 1 2 3 4 5 6 7 8)
```

So we can now allocate collections of processors and *join* them into larger collections but we still aren't building structured collections. As a natural step on from `join` we introduce the function `glue` which *glues* shaped paralations together along their *edges*. An edge is defined to be the collection of sites which for a given direction have no neighbour. In figure 3-5 we glue the `S` edge of `box` to the `W` edge of `xob`.

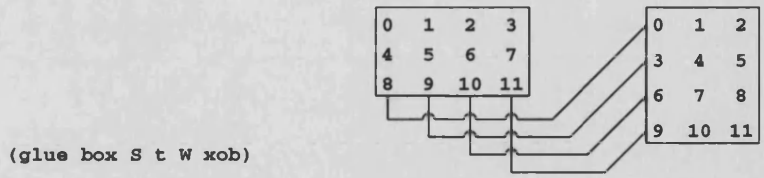


Figure 3-5: Gluing two rectangular paralations together

In this case we have two separate paralations so they are joined into a composite paralation. Then in each sub-paralation we have to replace the values in the structure slots with the index position of their new neighbours. To do this we enumerate the edges in each field and use this to create a mapping between the two paralations. This mapping is used to communicate the indices of the new neighbours to each other. The third argument to `glue` specifies if the edges should be glued in the *same* or *opposite sense*. Here they are glued in the same, *ascending index order*, sense.

Now when using `get`, where before we would receive the default value from the edges, where they have been glued together they are taken from the other paralation.

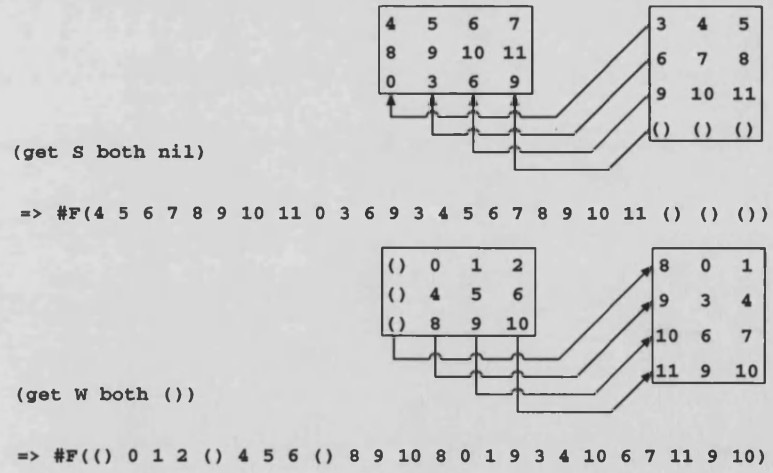


Figure 3-6: Moving data in a constructed paralation

`Glue` checks that its argument fields are from different paralations before joining them. If they are in the same paralation then it simply glues the edges of the paralation together. Figure 3-7 shows how rings and Moebius strips can be created by gluing the edges of rectangular paralations together.

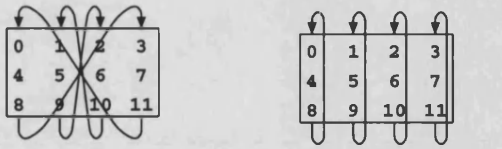


Figure 3-7: Gluing a paralation's edges together

Element-wise shape gives a much better handle on the connections between processors. With the functions `join` and `glue` we can construct paralations with complex internal connectivity defined by the construction process. However, although this gives us a way of describing the topology we are still having to allocate a complete set of new paralation sites for each constructive operation. It is also a rather coarse mechanism which operates on entire paralations and has a global view of the topology.

3.4 Shape Isn't Structure

In the previous three sections we have looked at various techniques used to give structure to a paralation. These methods are all based on creating a paralation which has some kind of *shape*: This is a useful enhancement to the Paralation Model but it does not constitute active memory programming.

Giving a paralation shape is a way of describing the site locality, i.e. where the sites of a paralation are in relation to each other. The shape access mechanisms give the paralation the appearance of the structure as well as the locality. So as well as moving data around the structure of the paralation we can reference the individual sites as if they were a part of the structure.

Locality is certainly an aspect of active data structures, if we connect two processors we are indicating they should be able to communicate easily. But there is more to an active data structure than the locality imposed on a paralation by a set of monolithic communication operations. Building a data structure should be relatively simple task, but defining the correct mappings can be complicated, particularly for any non-trivial structure, and it is often difficult to tell if the structure could have been better represented by a different set of mappings. As such, shaped paralations are best suited to regular structures. Further, manipulating data structures is difficult, a simple change to the connectivity means rebuilding the mappings while adding to the structure will mean reallocating the paralation and creating a new set of mappings.

The real problem is that most of this is simply cosmetic, and doesn't actually contribute to the Paralation model. All the mechanisms supplied can be implemented by the programmer, programmers

are after all usually well practised in the art of using one simple data structure (e.g. an array) to represent complex data structures (e.g. binary trees). The main contribution is a useful hook to allow the underlying architecture to be taken advantage of. But this aspect is invisible to the programmer, who is (rightly) unaware of it.

Paralation Views represent a more important contribution to the language since they permit a degree of control over paralation sites that was previously not possible. A part of a structure can now be extracted and operated on in isolation, whereas before it would have been necessary to move a portion of the paralation into a new paralation to do this. Thus we no longer need to allocate new paralations when part of an existing paralation can be used, and the cost of moving data into the new paralation is eliminated.

Paralation Views further enhance shape by allowing a paralation to have several shapes, and so we have a limited way of modifying the paralation's structure. But although we can have multiple structures we still cannot easily manipulate the structure: to do this still requires creating new mappings, or new paralations if we wish to add to the structure.

There are also aspects of views which are simply cosmetic, for example there is nothing to stop programmers implementing multiple shapes for paralations using mappings. In addition a `take`-like function can be implemented that does not require paralation sites to be shared. But not all aspects of views can be implemented using existing paralation primitives and the ability to share paralation sites represents a real contribution to the model.

The elementwise shape in Section 3.3 is very similar to the shaped paralations of Section 3.1: both view shape as a property of the entire paralation. But like Paralation Views it recognises the importance of the paralation's sites and the shape is defined on an individual site basis. The main advantage of this is it gives a simple hook for gluing structures together.

Being implemented in Paralation Lisp, elementwise shape does not extend the language (although it does represent a hook for the implementation to take advantage of the architecture). For this reason paralation sites cannot be reused and to glue paralations together a new paralation must be allocated (this was not the desired behaviour however).

Elementwise shape makes it much easier to manipulate the structure of a paralation, but it has various drawbacks. It is still oriented around entire paralations and so is a very coarse-grain mechanism. Neither is it very versatile, it is difficult to glue different types of shape together and is again only really suitable for regular structures. These limitations seem to be largely due to the *edge* concept which suits rectangles well, but is less applicable to an unbalanced binary tree.

Finally all three methods have one thing in common, the operations are unfamiliar and often both

verbose and confusing.

Although none of these methods constitutes active memory programming they have served to highlight some of the language requirements. And they have also given an indication of how some of the requirements can be met. Some of the key points are:

- **Familiar Constructs:** We would like to build active data structures using programming constructs similar to those already in common use.
- **Fine Grain Control:** Control should be at the site level rather than the paralation level. This should make the support of irregular, heterogeneous structures possible.
- **Scale Well for Large Structures:** Though we want site level control we should still be able to create large, regular structures easily.
- **Paralation Sites Sharable:** This reduces processor allocation and eliminates the unnecessary communication needed to move data into the modified structure.

In the next section we define a new set of extensions to Paralation Lisp in which we are able to create connections between sites in paralations explicitly. This gives a more local view of paralation topology and also eliminates the need to allocate new sites when constructing paralations.

3.5 Classified Paralations

In this section we present a class-oriented paradigm which gives greater control over the individual sites of a paralation. First we present some new communication technology which is used heavily by the active class system.

3.5.1 Targets

A *target* is a handle on a paralation site; we can think of this as an inter-processor pointer which simply points to a site, rather than an actual object *on* a site. A target can be created within an `elwise` expression by calling `make-target` with the index number of a site in the current paralation. Below we create a field where each element is a target pointing to the next site in the paralation:

```
(setq p (make-paralation 5))
⇒ #F(0 1 2 3 4)
(setq from (elwise (p) (make-target (+ p 1))))
⇒ #F(<target> <target> <target> <target> ())
```

Notice that in the case of the last element there is no site with index `5` and so `make-target` returns `nil`. These inter-site pointers can now be used to perform inter-site communication. The function `get` takes a field of targets and a data field from the same paralation:

`(get` *target-field data-field*`)`

The result is a field in the same paralation where each element contains the value on the site pointed to by the target on that element. So we can use our field of targets to shuffle a field's values left:

```
(get from p)
⇒ #F(1 2 3 4 ())
```

If an element of the target field is not actually a target then it is treated as a default value by the `get` operation. In the example above the last element of the field `from` is `nil` and this value is returned by `get`.

3.5.2 The Active Object System

The Active Object System, or TACOΣ, is an object-oriented extension to the paralation model which fulfils some of the requirements of an active memory language. Informally the requirements are the ability to manipulate processors with the same ease we currently manipulate memory.

In general, any class system is a high-level mechanism for building complex data structures in memory, hiding the details of memory management and pointer manipulation from the programmer. TACOΣ supplies a similar mechanism for creating active data structures, hiding the details of processor allocation and the creation of communication links.

In the same way the paralation model can be used to extend any base language the Active Class system should fit in with any existing object system in the language. The system is described here via the functionality of the EuLisp based implementation and reflects TEΛOΣ, the EuLisp object system. TEΛOΣ is described in full in the language definition [46], but much of the syntax is also similar to that used by CLOS, the Common Lisp Object System [19].

Defining Active Classes

An active class is defined using the `defactive-class` defining form which is similar to an ordinary `defclass`

`(defactive-class` *class-name* `(`*super-class**`)` `(`*slot-description**`)` *class-option**`)`

The syntax of `defactive-class` is as follows:

class-name ::= *symbol*

super-class ::= *class*

slot-description ::= `initarg` *symbol* |
`initform` *form* |
`reader` *reader-name* |
`writer` *writer-name* |
`accessor` *reader-name*

class-option ::= `constructor` *constructor-spec* |
`predicate` *constructor-name*

As an example we give below the definition of the active-class `plist`, not to be confused with Lisp "property lists", which will allow us to manipulate paralations as though they were lists. It has two slots called `pcar` and `pcdr` with reader functions of the same names and corresponding updator functions.

```
(defactive-class plist ()
  ((pcar accessor pcar
         initarg pcar
         initform (make-target (here)))
   (pcdr accessor pcdr
         initarg pcdr
         initform (make-target (+ (here) 1))))
  constructor (pcons pcar pcdr)
  predicate plistp)
```

Building Structured Paralations

When an active-class is instantiated we allocate a processing site which has a set of named slots associated with it. The easiest way to create an instance is using the constructor function defined in the `defactive-class` form. This allocates a single instance and sets the slots to the given arguments. In our `plist` definition we defined a constructor `pcons` which takes values for both `pcar` and `pcdr`.

```
(setq q (pcons 'last ()))
```

An instance of an active class is an active object. So far this is very similar to an ordinary object system. However here we have not simply created a class instance, but a single element paralation which has a structure defined by this class instance. The result of `pcons` is the index field for this new paralation:

```
⇒ #F(0)
```

Though the class instance is not apparent to us the `plist` accessor functions can be used to access its slots.

```
(pcar q)
⇒ last
```

If an argument given to the constructor is another active-class instance then this is interpreted as a communication link between the processing sites and the system packages up all the connected sites in a new paralation. Below we create an active list of three elements and in the `pcar` slot we store a symbol representing how far each element is from the end:

```
(setq p (pcons 'but-two (pcons 'but-one q)))
⇒ #F(0 1 2)
```

The result is a field which is *pointing* to the first element of the paralation, since that was the last element created. We need fields to identify a paralation site in this way so that accessor functions can be applied to paralations of more than one element:

```
(pcar p)
⇒ but-two
```

So far the TACOΣ object's behaviour matches our intuitive expectations of data structures. We have created objects and stored data in their slots which we have later retrieved. However, although we have built an active list in a familiar fashion it is not an ordinary data structure, it is a collection of processors which have structure associated with them. This difference becomes apparent when we apply `pcdr` to our active list:

```
(setq pcdr-of-p (pcdr p))
⇒ #F(0 1 2)
```

When p was created the `pcdr` slot was set to the result of another `pcons` operation. Thus intuitively we may have expected the result of this expression to be the two-element paralation representing an active list of two members. Rather than returning the two-element paralation we think of `pcdr` as returning another site in the same paralation. To give a handle on individual sites of paralations each field is thought of as *pointing* at a particular site of the paralation. In our example, the field p points at the first site in the paralation, this being the site returned by the final `pcons` operation. The result of applying `pcdr` to p is a field containing the same values but pointing at the second site in the paralation. In this way we can use the TACOΣ accessor functions to navigate round the structure of a paralation in a familiar fashion, for instance we can define an active `list-ref` function:

```
(defun plist-ref (plist index)
  (cond ((null plist) ()
        (zerop plist) (pcar plist)
        (t (plist-ref (pcdr plist) (- index 1))))))
```

So far the TACOΣ accessors only allow us to access data held in the slots of the TACOΣ objects. Each paralation has data associated with it in the form of fields, and we may want to access this data via the paralation structure as well. For example we can load some data onto our active list paralation:

```
(setq p-data (elwise (p) (list-ref '(a b c) p)))
⇒ #F(a b c)
```

And we may want to know what element of `p-data` is associated with the second element of the active list. The function `value` and its updator access the elements that are pointed at by fields. Below we access the second and first elements of `p-data` and then update the second element.

```
(value p-data)
⇒ a
(value (pcdr p-data))
⇒ b
((setter value) (pcdr p-data) 'Woah)
⇒ #F(a Woah c)
```

A structured paralation then is a set of processors where the nodes have class, hence a classified paralation. The slots of the classes point to other sites in the paralation and this gives the structure. A field in a structured paralation also *points* at a paralation site and this allows the structure to be traversed using the active-class accessor functions. Because the paralation need not be homogeneous it is useful to be able to determine the class of the paralation site currently being pointed at:

```
(active-class-of pcdr-of-p)
⇒ <plist>
```

we can also use the predicate function created by the active-class definition:

```
(plistp pcdr-of-p)
⇒ t
```

Communication in Classified Paralations

In TACOΣ the active objects represent abstract processors, which have multiple communication links and some processing capability. The active objects are used as the processing sites of paralations,

a field in the paralation represents data stored on these abstract processors and `elwise` is used to execute code on them.

Like an ordinary data structure, an active data structure is made up of nodes and connections, but rather than having a process *walk* over the structure we have communicating processes associated with each node. Although we can activate a process at each node in the structure using `elwise` the structure of the abstract processors is not actually apparent to the processes. We introduce a new function `structure` which returns the TACOΣ object a process is "executing" on. For completeness we have:

```
(structure)
⇒ #<mp-host>
```

However `structure` is only of any real use within the body of an `elwise` statement:

```
(elwise (p-data) (structure))
⇒ #F(#<plist> #<plist> #<plist>)
```

When we apply any of the readers, writers etc. of TACOΣ to a field, they are applied to the active-class instance on the site the field points to, so in the example above `p` was pointing at element 0 and `pcdr` returned a field pointing at the second element. These active-class functions can also be applied directly to the active-class instance returned by `structure`. So within the body of an `elwise` statement we can access the active-class instance associated with each paralation site. Thus:

```
(elwise (p) (pcar (structure)))
⇒ #F(but-two but-one last)

(elwise (p) (pcdr (structure)))
⇒ #F(<target> <target> ())
```

Though the `pcar` slot of each `plist` instance contains the data we would expect we can see something different has happened with the `pcdr` slots. These are the slots we were using to define the structure of the sites in our paralation, rather than placing the field in the slot, a target for the site has been created and stored in the slots. Using the function `get` defined in section 3.5.1 we can move data around a paralation using the internal structure defined by the TACOΣ objects:

```
(get (elwise (p) (pcdr (structure))) p-data)
⇒ #F(Woah c ())
```

An important advantage of the active-class system is that the structure and hence the communication patterns can be changed very easily and do not require the entire paralation to be reallocated.

For example it is straightforward to make our simple `plist` into a circular `plist`.

```
((setter pcdr) (pcdr (pcdr p)) p)
⇒ #F(0 1 2)

(get (elwise (p) (pcdr (structure))) p-data)
⇒ #F(Woah c a)
```

Notice that modifying the structure affects all the fields in the same paralation. To understand why this happens we must quickly review the organisation of paralations and fields. A paralation is a collection of processing sites, and we now think of these sites as being instances of active classes. So making changes to the active class instances effectively changes the structure of the paralation. The paralation itself is never directly visible to us and must be accessed via the fields that belong to it. The fields `p` and `p-data` belong to the same paralation, so the changes made via `p` are reflected in `p-data`.

The paralation's structure can also be modified by accessing the TACOΣ object within an `elwise` expression. For example suppose we want the `pcdr` to point at the next but one element:

```
(let ((next (elwise (p) (pcdr (structure)))))
  (elwise ((next-next (get next next)))
    ((setter pcdr) (structure) next-next)))
⇒ #F(<plist> <plist> <plist>)

(get (elwise (p) (pcdr (structure))) p-data)
⇒ #F(c a Woah)
```

Because the communication pattern is selected using an `elwise` expression it is simple to move data round a heterogeneous data structure. We can imagine collating data on a network where the link that each node should read depends on its active-class. It is straight forward to access the appropriate slot based on the active-class of the site and the resulting field can then be passed to `get`.

Creating Classified Paralations

Although the active-class constructor functions give us a natural way of creating structured paralations it is a lengthy and tedious way of creating very large structures, especially if they are regular. For example with our `plist` example, if we are creating a simple active list of n elements then for each element i the `pcdr` points to element $i + 1$.

We extend the syntax of `make-paralation` to allow an active-class to be specified, each site of the resulting paralation will be an instance of this active-class:

`(make-paralation` *size active-class init-option**`)`

When the active-class instances are created their slot values will be the result of the corresponding active-class initform expressions. We can use `make-target` in the `initform` expressions to define inter-site connections, this gives a straightforward way of creating large collections of processors which have uniform structure rather than allocating and connecting all the sites individually. So that we can create non-trivial structures we introduce the function `here` which returns the index of the processing site it is executed on — again this is of little use outside parallel expressions. The initform for `pcdr` is:

```
(make-target (+ (here) 1))
```

This initialises the `pcdr` slot of each `plist` instance to a target pointing at the next site in the paralation. We can now create a 5 element `plist` with a single expression:

```
(setq p (make-paralation 5 plist))
⇒ #F(0 1 2 3 4)

(get (elwise (p) (pcdr (structure))) p)
⇒ #F(1 2 3 4 ())
```

Often the initialising expressions require additional information; for example if trying to create a grid-shaped paralation we will need to know the width of the grid:

```
(defactive-class grid ()
  ((up
    reader up
    initform (make-target (- (here) width)))
   (down
    reader down
    initform (make-target (+ (here) width))))
  predicate gridp)
```

This definition of `grid` which defines up and down connections for each site contains an unresolved variable `width`. The `make-paralation` *init-option* allows values for these variables to be specified. An *init-option* is a symbol followed by the corresponding value. So to create a 3×4 grid:

```
(setq 3-by-4 (make-paralation (* 3 4) grid 'width 4))
⇒ #F(0 1 2 3 4 5 6 7 8 9 10 11 12)

(get (elwise (3-by-4) (down (structure))) 3-by-4)
⇒ #F(4 5 6 7 8 9 10 11 () () () ())
```

Although this could be done by additional `elwise` expressions this is a useful extension making the creation of classified paralations much cleaner.

Modifying Structures

When building an active data structure using the constructor functions the active-class system collects all the connected sites into a single paralation. In fact, it assumes that the paralations involved *are* themselves connected and simply takes the union of them — a relatively cheap operation.

However we can modify the structure further using the active-class accessor functions and this may add or remove sites from the structure. In this case only the structure is modified and the system does not attempt to generate the connected paralation for each operation. This means the operations are cheaper to use but the resulting data structures may not be contained in a single paralation. Further the paralations may no longer be connected, although having redundant nodes in a paralation is not necessarily a problem, it can be messy and these nodes could be garbage collected for later reallocation.

The active-class system supplies some additional functions to address these problems. The function `connected` takes a field and creates a new paralation containing all the connected sites in the structure pointed at by the field. The resulting field contains the elements of the argument field which are in the new paralation. Because the active-class instances represent individual processing sites they can each occur only once in the result paralation. This means it should be safe to use `connected` with any kind of cyclic active data structure. An implementation of `connected` is briefly discussed in Section 6.3.3.

Returning to our `plist` example we may wish to extract the last three elements of our 5 element list. Generating the set connected to the third element has this effect:

```
(setq last-three (connected (pcdr (pcdr p))))
⇒ #F(2 3 4)
```

As it is likely we will want to use values from fields in paralations which contributed to the new paralation we also supply the function `project`. This performs a task similar to that done by `take` in Paralation Views (Section 3.2.2).

(`project` *destination-field data-field default*)

The result of `project` is a field in the same paralation as the *destination-field* containing the elements of the *data-field* which are on sites in the *destination-field*.

```
(setq c-to-e (project last-three
                      (elwise (p)
                        (list-ref '(a b c d e) p))
                      ()))
⇒ #F(c d e)
```

If a site in the destination field doesn't have a value in the data field then that element is given the *default* value. To illustrate this we can reverse the last `projection`. In this case there are two sites in the paralation that field `p` belongs to that do not have values in the field `c-to-e`.

```
(project p c-to-e '*nothing*)
⇒ #F(*nothing* *nothing* c d e)
```

Comments

The dual nature of classified paralations can at first be confusing. On the one hand we have something that appears to be an ordinary data-structure which can be worked with in familiar ways. On the other we have a collection of processing sites with some communication patterns defined on them.

If we consider the way we use ordinary data structures we usually have a collection of nodes with memory pointers connecting them, calculations using such a structure usually require a single process to walk over the structure performing individual calculations at each node. Clearly many problems can be solved more quickly by having a processor at each node of the structure but how these processors become active remains a problem as we usually only have a handle on one *root* node in the structure. An obvious solution is to propagate an activation wave through the structure from some root node. However for some structures, like lists, this activation wave is a linear process causing the parallelism to degenerate to serial behaviour. TACOΣ addresses this problem by parcelling up all the nodes in an active structure into a single paralation, each node can then be activated simultaneously using one operation, i.e. `elwise`.

The process of propagating a wave through the data structure is in fact what the function `connected` does to find the members of the new paralation. Although this is as expensive as propagating a process activation wave through the active structure it need only be done once, the sites

of the structure can then be activated simultaneously for all parallel operations there after until the structure is modified again.

3.5.3 Some Alternative Semantics

So far we have stuck with the Paralation Model's philosophy of separating computation and communication which it does by having distinct mechanisms for each. Thus `elwise` serves as a bulk synchronisation operation for computation between communication phases when inter-site dependencies could become an issue. The description of TACOΣ maintains this organisation by supplying an additional communication function, `get`, which also operates on entire fields. However this organisation is at odds with an object-oriented approach to active memory programming, this is apparent if we consider how objects are typically used:

Operations on a class instance often involve checking the class, extracting the values of some slots, performing some computation and possibly setting some slot. However for an active-class instance we will have to factor the slot accesses, since these are potentially communication, out of the operation and then give the results as additional arguments to another expression. This is not a difficult task but it forces the programmer to consider communication as monolithic data permutations rather than accesses between active objects and so it somewhat compromises TACOΣ as an object oriented active memory programming paradigm.

Further, in this chapter, we have seen the emphasis being placed on the individual sites rather than the entire paralation. Thus it seems quite natural to make `get` an elementwise operation rather than a fieldwise operation. Then we will be able to write object-oriented functions and apply them in parallel, rather than parallel functions oriented around collections of objects. For example, the function to calculate the average value on the four neighbours of a `grid` instance has a straightforward definition:

```
(defun average (value)
  (if (not (gridp)) (error "Invalid active-class" bad-class)
    (/ (+ (get N value)
          (get E value)
          (get S value)
          (get W value)) 4.0)))
```

And could now be used in an `elwise` expression:

```
(elwise (intensity) (average intensity))
```

However although the interpretation of `get` was simple when applied to an entire field it is less obvious what is happening when it appears in the body of an `elwise` expression.

To achieve elementwise interprocessor communication we consider each TACOΣ object to have a *visible* location which is readable by other processors. Objects can be stored in this location using the function `update`:

`(update` *value-obj*`)`

In the same way that the object returned by `structure` (see Section 3.5.2) is implicit, so is the location accessed by `update`. Again, this is motivated by considering each paralation site to be a TACOΣ instance: thus `update` simply accesses the *visible* location of the TACOΣ instance it is executing on and only the new value needs to be given.

Processors can read, that is take a copy of, the contents of the visible location on a remote processor using `ref`. This is a non-destructive read, so the value may be read by several processors.

`(ref` *target-obj*`)`

For `ref` only the inter-site pointer needs to be specified as this implicitly defines which TACOΣ instance, and hence, which *visible* location to access. In the event that *target-obj* is not a target, `ref` simply returns the object *target-obj*. As we consider the function `get` to be a TACOΣ operator we define it to accept a TACOΣ reader rather than a target.

`(get` *tacos-reader value-obj*`)`

Below we define `get` in terms of `ref` and `update`. First `value-obj` is stored in the *visible* location, so this is the object that other processors will `get` from this site. The TACOΣ reader is then applied to the local TACOΣ object and the result passed to `ref`. Thus when `get` is used in parallel we think of each participating processor as contributing a value and returning a remote value or simply the contents of the specified slot.

```
(defun get (tacos-reader value-obj)
  (update value-obj)
  (ref (tacos-reader (structure))))
```

This definition is similar to the behaviour of the communication functions supplied by TUPLE (see Section 2.1.2). An important difference is the contents of the visible location of each processor are persistent and so they may be read from even if the processor itself is not active.

This naturally raises various problems of synchronisation. However if we assume we are restricting ourselves to SIMD architectures then synchronisation will not be a problem. This goes

against the grain of the Paralation Model which is supposed to be an architecture-independent parallel programming model, however this is an inevitable effect of specialising a language for a particular architecture. This need not be seen as a problem as we are using the paralation model as the basis for an active memory language, not creating an active memory language that fits in with the Paralation Model.

Throughout this description the TACOΣ functions have all been applied to the object returned by the function `structure`. This was primarily to help clarify the explanation of the operations. Because in the same way that the object returned by `structure` is implicit in its execution context, so too is the object that should be the argument of a TACOΣ function. This means we can arrange for the TACOΣ functions to use this object automatically and `structure` can be removed from the definition.

There are two reasons why we may want to do this, the first is it simplifies the code a little and for this reason we will use this convention from here on. The second is that during the original design stages it was felt unwise to give the programmer access to the objects themselves. The possible control over the active objects seemed too broad to be effectively supported. It also seemed likely that this much power would be too dangerous to put into the hands of programmers, for them and their users. As the design matured and we were able to experiment with the language it now seems that access to the objects would be highly desirable. This represents an important area for future work and is discussed further in Section 7.1.3.

Micro-Macro Equivalence

By making `get` an elementwise operation we have broken with another Paralation Lisp philosophy; that execution and communication should be rigidly separated [55, Ch. 3]. To this end the communication functions, `match` and `move`, operate on entire paralations and combining collisions is the only mixing of communication and execution that occurs. Again this does not actually constitute a problem as the Paralation Model is only the basis of our active memory language. Further, the key reason for the strict separation is to facilitate the *Architectural Independence* of Paralation Lisp by making it easy to implement for unsynchronised parallel architectures. We however, are primarily interested in synchronised data-parallel architectures, and in addition to this, much progress has now been made in the efficient compilation of data-parallel programs for MIMD architectures [28, 27, 16, 17]. Hence we do not share all the goals of Paralation Lisp and there are also alternative ways of achieving them.

But, most importantly, the removal of the restriction is a valuable enhancement to the expressiveness of the language. The justification for making `get` an elementwise operation was our desire to

write active object-oriented code, i.e. to be able to define functions which can be applied to objects in parallel and perform both computation and communication.

These requirements are almost identical to the behaviour of a language that is micro-macro equivalent (see Section 2.3.2). In such a language there are two views of computation, one of a single program manipulating a collection of data (a processor of collections) and another of a collection of processors manipulating their individual data (a collection of processors). If there is an equivalence between the two views then we can program in the small, that is for an individual processor, and then scale to the problem size.

Thus, by making communication possible *during* parallel execution we are no longer forced to consider the entire collection of processors. This means we can write code for individual TACOΣ objects handling both communication and execution, i.e. object-oriented code. This code can then be applied to all the instances of the objects we have, in parallel.

So though we have broken away from Paralation Lisp, we have improved the micro-macro equivalence of the language in the process, which is known to be a useful property [61]. Further, without this micro-macro equivalence the utility of active objects becomes very restricted and we cannot easily use an object-oriented style of active memory programming.

3.6 Summary

In this chapter we have looked at ways of extending the Paralation Model so that it meets the requirements of active memory programming.

We first looked at some existing extensions that give structure to a paralation by defining locality within the paralation, i.e. which sites are near each other. Locality is certainly an aspect of active data structures: connected processors in a structure can communicate easily as can processors that are near each other. Although *shaped* paralations enhance the language a paralation with structure does not give us all the utility we expect of an active data structure. Structures can't be built, they are defined as a global property of the paralation, and this also makes it difficult to modify the structure. With Paralation Views shape becomes more versatile, paralations can have multiple shapes and can also be efficiently decomposed into sub-paralations. So although we still cannot construct paralations we can take them apart, and some structural modifications can be supported by multiple shapes. Elementwise shape has much the same limitations, but by defining its locality on a per-site basis it creates a useful hook for connecting shaped paralations together. But because control is at the paralation level it is very coarse and still best suited to simple, regular structures like rectangles.

Having looked at these systems we then introduced The Active Object System, TACOΣ. This

applies the ideas of typical object systems to active memory programming, in the same way an object system hides the details of memory allocation and pointer construction, TACOΣ hides the details of processor allocation and the construction of communication links. In this way, the control of processors and communication is supplied through a familiar mechanism, which encapsulates both locality and access. Control is at the paralation site level, these are considered to be the *active* objects, this gives fine control over the structure of paralations allowing individual sites and links to be created and modified. In addition, the objects can be created and manipulated in parallel, this means that many structures too large to be built site by site, are still practical TACOΣ structures. It also proved necessary to relax one of the Paralation Model's constraints, i.e. the strict separation of communication and computation. By doing this we give the language the necessary micro-macro equivalence needed to be able to write code in an object oriented fashion. This means as well as being used to build paralations, TACOΣ can be used effectively in the code executed on the paralations.

So it seems that active objects fulfil many of the requirements of active memory. However, although TACOΣ goes further than existing extensions we have yet to see if it is actually useful. To this end the next chapter looks at various examples where TACOΣ is used both to build active data structures and in the code executed on them.

Chapter 4

Using Active Objects

Having characterised the active memory architecture in Chapter 1, we saw in Chapter 2 the languages for these machines, such as NESL and Paralation Lisp, are not really active memory programming languages. Essentially they are data parallel languages which give good control over the computers but do not embody the ideas we are interested in. In Chapter 3 the ideas in ordinary object systems were used to define the Active Object System as an active memory extension to the Paralation Model. Although the design always considers the requirements of the object oriented programmer we have yet to see if it is actually usable, and if so whether it is useful.

In this chapter we look at a variety of problems and their solutions using TACOΣ. These will illustrate how the programming style promoted by TACOΣ is much more object-oriented than that of straight Paralation Lisp. Some of the examples will also motivate more modifications to Paralation Lisp which will allow TACOΣ to be used more effectively.

4.1 Parallel Prefix

Computing all the partial sums of an array is often referred to as a "sum-prefix" operation, because it computes sums over all the prefixes of the array. This is also referred to as a scan operator: for example plus-scan (+/) in APL and `+-scan` in NESL. Prefix-sum can be generalised from summation to any associative combining operator, obvious examples are product, logical or, logical and, minimum and maximum. On massively parallel architectures, where each element of the array can be stored on a separate processor a general prefix operation can be implemented so that its complexity is $O(log_2 n)$.

An implementation of parallel prefix that is easy to understand uses pointer doubling. Each processor has a pointer to the next processor which we call its *buddy*. Each processor that has a buddy reads its buddy's value and its buddy's buddy. The processor combines its buddy's value with

its own to find a new value for its buddy. It then sets its own buddy, to its buddy's buddy. This is repeated until none of the processors has a buddy, when the prefix operation will be complete. On each iteration the length of the pointers is doubled, hence the complexity is $O(log_2 n)$. Below is the definition of this algorithm given by Steele and Hillis. This is expressed in terms of arrays of values where the indices of the arrays are used as inter-site pointers. The symbol $\odot$ is used to represent a general, associative combining function.

for all k **in parallel do**
  $buddy[k] := next[k]$
  while $buddy[k] \neq null$ **do**
    $value[buddy[k]] := value[k] \odot value[buddy[k]]$
    $buddy[k] := buddy[buddy[k]]$
  od
od

This is a very brief and clear description of the algorithm and it is fairly obvious how data is being moved in the collection of sites. However we cannot directly implement this algorithm in a parallel functional language. The chief difficulty is in the expression $value[buddy[k]] := \ldots$, which updates *value* on the buddy processor. This is simply an inter-processor write, however the processor being written to may be inactive as this is determined by the enclosing **while** statement. In a functional language *value* will be a binding, not simply a memory location with an address, so it is difficult to update *value* on a remote processor without the cooperation of the target processor. This problem is quite clearly manifested in the functionality of TUPLE's communication primitives (see Section 2.1.2), which can only communicate with other active processors, otherwise a default value is returned.

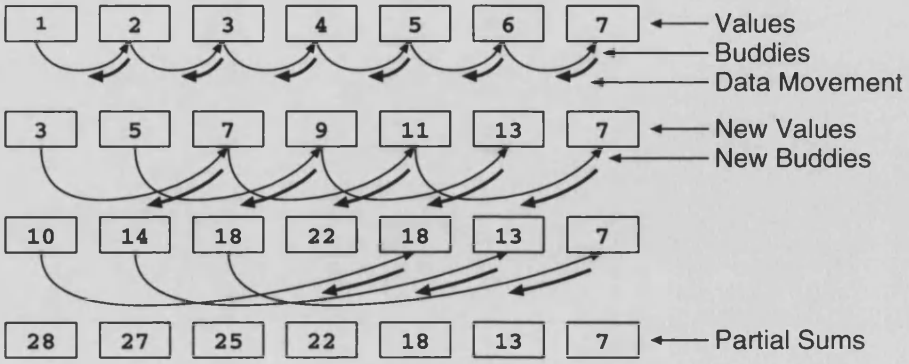


Figure 4-1: Data Movement In Parallel Prefix Sum Operation

It is quite simple to modify the algorithm so that it can be implemented within these constraints.

Firstly if the direction of the prefix operation is changed we only need to read from the remote processors. This means we do not need to update remote bindings and also we only require a single read, rather than a read and a write as before. Secondly we keep all the processors active so that they can all participate in the communication phase by modifying the condition to **while** and adding an extra conditional within its body.

for all k **in parallel do**
  $buddy[k] := next[k]$
  while $\exists k : buddy[k] \neq null$ **do**
    $buddys\text{-}data[k] := data[buddy[k]]$
    if $buddy[k] \neq null$
      $data[k] := data[k] \odot buddys\text{-}data[k]$
      $buddy[k] := buddy[buddy[k]]$
    fi
  od
od

In this version all the sites are active during the communication phase, but only those for which $buddy[k] \neq null$ perform the computations. Figure 4-1 shows the stages of a small prefix-sum operation based on this algorithm. This version of the algorithm can be implemented in TACOΣ in a way that reflects the feel of the algorithm closely. We define an active class with a `next` slot set appropriately and a `buddy` slot as used in the algorithm:

```
(defactive-class sequence ()
  ((next reader next
         initform (make-target (+ (here) 1)))
   (buddy accessor buddy)))
```

To implement $\exists k : buddy[k] \neq null$ we reduce a field composed of each site's `buddy` using `vref` with `or` as the combining function, if this returns `t` then there are still sites with non-`nil` buddies. The rest of the code needs little explanation:

```
(defun parallel-prefix (data comb)
  (elwise (data) ((setter buddy) (next)))          ;buddy[k] := next[k]
  (while (vref (elwise (data) (buddy)) or)         ;∃ k : buddy[k] ≠ null
    (elwise (data)
      (let ((buddys-data (get buddy data)))        ;data[buddy[k]]
```

```
      (when (buddy)                                    ;buddy[k] ≠ null
        ((setter buddy) (get buddy (buddy)))           ;update buddy
        (setq data (comb data buddys-data))))))        ;combine data
  data)
```

Below is an implementation of pointer doubling based on one given by Gary Sabot: this version has been converted to EuLisp. On each iteration, a fresh mapping must be made to move the data and the mapping is created in terms of the index field and the desired length of the pointers. The algorithm terminates when the length of the *buddy* pointers is greater than the size of the paralation.

```
(defun parallel-prefix (comb data)
  (let ((psize (length data))
        (order (index data))
        (data (elwise (data) data))
        (distance 1))
    (while (> distance psize)
      (let ((shifted-data (move data (match order (elwise (order)
                                                    (+ order distance)))
                                () 'no-data)))
        (elwise (data shifted-data)
          (if (eq shifted-data 'no-data) ()
            (setq data (comb shifted-data data))))
        (setq distance (* 2 distance))))
    data))
```

If we ignore the cost of creating mappings for each level (using TACOΣ the targets are only created once) this implementation has the same complexity as the TACOΣ based version, which was described above. However it does not really resemble the algorithm, nor does it have its elegance.

The buddy method for calculating a parallel prefix is a good example for TACOΣ because it is an active object-oriented algorithm: it is based on an active data structure, a linked list of processors, and the code describes the behaviour of a single node, which is then applied to the entire list in parallel. For these reasons the buddy algorithm can be implemented with TACOΣ with minor modifications. However it maps poorly into Paralation Lisp which, as we have discussed earlier, does not encapsulate active memory programming well.

We also see the utility of being able to move inter-site pointers around, a quite natural operation for an object-oriented programmer which is tortuous in Paralation Lisp. The $buddy[k] := buddy[buddy[k]]$

expression requires a single read with TACOΣ (as it should) but with mappings `match` must be used again each time to create the correct inter-site pointers.

However the TACOΣ solution does still have its weaknesses. Even with communication inside the body of `elwise`, and modifying the algorithm to remove inter-processor writes, the TACOΣ code is still not entirely object-oriented. The check $\exists k : buddy[k] \neq null$ must be executed outside the body of an elwise, this breaks the modularity somewhat. But we should remember that this is a feature inherent in the paralation model as `elwise` is a bulk synchronisation operator [69].

4.1.1 Scans and Active Objects

Because the buddy algorithm is based on the connectivity of the participating sites it can be used meaningfully with a variety of active objects. Matrices and vectors present a good example of this utility. To illustrate this we will look at a common matrix operation, multiplying a vector by a matrix. To do this we will define a TACOΣ object for a matrix site where the rows and columns are connected, i.e. each element is connected to the elements lying to the south and west of it. We will also find it useful to store row and column data in the TACOΣ object:

```
(defactive-class matrix ()
  ((row reader row
        initform (make-target (/ (here) width)))
   (col reader col
        initform (make-target (remainder (here) width)))
   (up reader up
       initform (make-target (+ (here) width)))
   (left reader left
         initform (if (= (remainder (here) width) (- width 1)) ()
                      (make-target (+ (here) 1))))))
(defun make-matrix (rows cols . data)
  (let ((mat (make-paralation (* rows cols) matrix 'width cols)))
    (if data (elwise (mat) (list-ref data mat))
      mat)))
```

In order to multiply a vector by a matrix (assuming their dimensions match) we need to spread the vector across the matrix, that is each column of the matrix must have a copy of the vector. For example consider multiplying a 3-element vector by a 3×4 matrix (we represent a vector by a $1 \times n$ matrix):

```
(setq mat (make-matrix 3 4 5 6 4 3 11 5 7 4 7 7 9 7))
⇒ #F( 5 6 4 3
      11 5 7 4
       7 7 9 7)
(setq vec (make-matrix 1 3 4 5 1))
⇒ #F(4 5 1)
```

One simple way of spreading a vector across the rows of a matrix is to dereference the vector in parallel, that is for each site in the matrix to access the appropriate element in the vector using `field-ref`.

```
(elwise ((mat (make-matrix 3 4)))
  (field-ref vec (row)))
⇒ #F(4 4 4 4
      5 5 5 5
      1 1 1 1)
```

This is fairly neat solution but it may be inefficient. Each `field-ref` requires inter-processor communication, with four processors trying to access the same element. Typically the hardware will have to sequentialise the four accesses [38, page 2-29] and so the operation will have complexity $O(\texttt{cols})$, but using a prefix operation the complexity will be $O(\log_2 \texttt{cols})$. We define a general scan operator which allows the direction of the scan to be specified by giving the appropriate TACOΣ accessor. This effectively specifies the TACOΣ instance slot to be used as the buddy slot in the algorithm. The initial contents of the *buddy* slot, hopefully a target, are preserved in `real-buddy` and restored when the scan is complete using the `unwind-protect` form.

```
(defun scan (data comb buddy)
  (let ((real-buddy (elwise (data) (buddy))))          ;store buddy's value
    (unwind-protect
     (while (vref (elwise (data) (buddy)) or)
       (elwise (data)
         (let ((buddys-data (get buddy data)))
           (when (buddy)
             ((setter buddy) (get buddy (buddy)))
             (setq data (comb data buddys-data))))))
     (elwise (real-buddy) ((setter buddy) real-buddy)))
    data))                                             ;restore buddy
```

Now the vector can be duplicated across the matrix's columns by reading it into the first column using field-ref and then spreading it across the matrix using `scan`. To do this we specify that each processor's buddy is its `left` neighbour and the combining function returns the buddy's value. Thus the final values will all have been read from the left-most column. By doing this the complexity of spreading the vector is $O(\log_2 \texttt{cols})$ — as the complexity of the `field-ref` phase is only $O(1)$ since there will be no collisions.

```
(setq dup-vec (scan (elwise (mat)
                      (if (= (col) 0) (field-ref vec (row)) ()))
                    (lambda (data buddys-data) buddys-data)
                    left))
⇒ #F(4 4 4 4
      5 5 5 5
      1 1 1 1)
```

The next stage, computing the products at each site is straightforward. The elements of the result vector are given by summing over these columns of products, since the rows of the matrix are connected we can use `scan` again to produce the sums. The result vector is located in the last row of the resulting field.

```
(setq tmp (scan (elwise (dup-vec mat) (* dup-vec mat)) + down))
⇒ #F(20 24 16 12
      75 49 51 32
      82 56 60 39)
```

All that remains to be done is to extract the result into a new vector, we create a vector paralation of the appropriate size with a field made up from the values in the last row of the field `tmp`.

```
(elwise ((rvec (make-matrix 1 (/ (length mat) (length vec)))))
  (field-ref tmp (+ (- (length mat) (length rvec)) rvec)))
⇒ #F(82 56 60 39)
```

Matrix multiplication is a useful example as it shows the utility of scan operations and how elegant and efficient solutions can be derived using scan and active objects. Another important point is that we are able to operate on rows and columns with equal ease. This is possible because the necessary information is held within the objects and the code is written for a single object so that its behaviour varies to suit the properties of the object. With ordinary Paralation Lisp we could represent a matrix as a field of fields, where each sub-field is a row of the matrix. This makes row based operations

easy but operations on columns are difficult as the representation will generally have to be changed. Paralation Views of course do allow us to view the matrix as both a collection of rows and a collection of columns.

A final point which is worth mentioning is that in the context of hardware, spread operations are often more efficient than inter-processor references: we saw this earlier when spreading a vector across the matrix paralation. Another example of this is broadcasts in nested parallel expressions. At the hardware level a single value can be broadcast to a set of processors in a single operation. Normalising a set of values is an operation that requires a broadcast, the maximum value must be identified and then communicated to all the processors.

```
(setq data (elwise ((i (make-paralation 5)))
             (list-ref '(3.9 8.5 9.8 7.2 5.7) i)))
⇒ #F(3.9 8.5 9.8 7.2 5.7)

(let ((max-val (vref vec max)))
  (elwise (data) (/ data max-val)))
⇒ #F(0.4 0.7 1.0 0.7 0.6)
```

In this expression it will be necessary for the value of `max-val` to be broadcast to all the sites in `data`'s paralation. This is not a difficult operation for most parallel architectures, but it becomes less easy when we are dealing with nested expressions. If we wish to perform the same operation for each field in a nested field then in general the value to broadcast will be different for each sub-paralation. As all the sites will typically share the same physical controller this cannot be done simultaneously and each value will have to be broadcast in turn.

The same operation can be achieved using two scans, one to identify the maximum value and another to spread it back to all the elements in the paralation. To do this we will need our paralation sites to be connected to their next and previous neighbours:

```
(defactive-class sequence ()
  ((next
    reader next
    initform (make-target (+ (here) 1)))
   (prev
    reader prev
    initform (make-target (- (here) 1)))))
```

Using our earlier definition of `scan` we can normalise `data` without the need for a broadcast:

```
(let ((max-val (scan (scan data max next) max prev)))
  (elwise (data max-val) (/ data max-val)))
⇒ #F(0.4 0.7 1.0 0.7 0.6)
```

In this case the process of communicating the maximum element to the other sites of a paralation is done entirely within the paralation and depends on its own connectivity. So if this expression is used for a nested field there will be no interference between the sub-fields and so no overhead. Of course the complexity of this operation is now proportional to the *log* of the largest field and this may actually be greater than the size of the parent field, in which case a sequence of broadcasts would be more efficient.

4.2 Gaussian Elimination

In the previous section we discussed and defined various machinery useful for operations on matrices. In this section we will try to use this machinery in another, more involved, matrix operation. Gaussian elimination is a method for solving systems of n linear equations in n variables of the form:

$$a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{in-1}x_{n-1} = b_i \text{ for } i = 1, n$$

The system can be represented using a matrix A of the coefficients and a vector b containing the right hand side of each equation, i.e. $Ax = b$. This equation can then be solved by reducing the matrix A to upper triangular form, where all the values below the leading diagonal are zero. Gaussian elimination is a simple method of reducing a matrix to upper triangular form.

Any transformations made to A must also be reflected in b. For simplicity we place both A and b in a single $n \times (n+1)$ matrix $A : b$. A will refer to this combined matrix from here on. The algorithm for Gaussian elimination is given below. It does not go on to find the solution using back propagation, though this is straightforward. The algorithm makes use of pivoting to help reduce the arithmetic error: this requires rows of A to be swapped. For a complete solution these swaps must be recorded but the details are omitted here. Neither do we check that the system has a solution, i.e. that A is non-singular.

The output matrix is found by successively modifying A with a sequence of steps that are executed for each index i from 0 to $n-2$. The steps are given below (note r and k are always integers).

1. **Max-row:** Find (*max-val, row*) such that: $\forall k \in [i, n) : |A[k][i]| \leq |A[row][i]|$

   That is, in column i the greatest absolute value on or below the diagonal is *max-val* on row *row*.

2. **Swap:** Swap rows i and row of matrix A.

   Modify $A \rightarrow A'$ such that: $\forall k \in [0, n] : A'[row][k] = A[i][k], A'[i][k] = A[row][k]$

3. **Normalise:** Modify $A \rightarrow A'$ such that: $\forall k \in [i, n) : A'[k][i] = A[k][i]/max\text{-}val$

   That is, divide all elements in row i by *max-val*, the element on the diagonal will now be 1.

4. **Update:** $A \rightarrow A'$ such that: $\forall r \in (i, n), k \in [i, n] : A'[r][k] = A[r][k] - A[r][i] * A[i][k]$

   This sets all elements below the diagonal in column i to zero.

```
bigger(a b)
  if |car(a)| > |car(b)|
    return a
  else
    return b
  fi

for i := 0 to n − 2 do                                        Max-Row
  for all row, col in parallel do
    if (col = i) ∧ (row ≥ i)
      contender[row][col] := cons(A[row][col] row)
    else
      contender[row][col] := cons(0 row)
    fi
  done
  max-row:= reduce(contender bigger)

  for all row, col in parallel do
    if (row = cdr(max-row))                                   Swap
      tmp[row][col] := A[row][col]
      A[row][col] := A[i][col]
      A[i][col] := A[row][col]
    fi

    if (row = i)                                              Normalise
      A[row][col] := A[row][col]/car(max-row)
    fi

    if (row > i) ∧ (col >= i)                                 Update
      A[row][col] := A[row][col] − A[i][col] * A[row][i]
    fi
  od
od
```

Figure 4-2: Pseudo-code for Parallel Gaussian Elimination

The algorithm is given in Figure 4-2 in an extended version of the pseudo-code used by Hillis and

Steele. The algorithm is fairly simple but can bear some explanation. In the **Max-Row** phase each site creates a *contender* for max-row, which is a *value*, *row* number pair. If the site is in column i and lies on or below the diagonal then it uses the matrix value for that site, otherwise a *no-hoper* contender is created where the value is zero. These pairs are reduced using the combining function *bigger* which chooses the pair with the largest absolute *value* each time, this gives the largest absolute value for the column and the row it lies on. The rest of the algorithm is straightforward, each phase activates an appropriate set of processors which update values and perform inter-processor communication.

Below we give an implementation of Gaussian elimination using TACOΣ. The code is based on the `matrix` active class defined in section 4.1.1. Some of the code is essentially identical to the algorithm, for example `max-row`:

```
(defun max-row (A i)
  (vref (elwise (A) (cons (if (and (= (col) i) (>= (row) i)) A 0) (row)))
        (lambda (a b) (if (> (abs (car a)) (abs (car b))) a b))))
```

We cannot implement **swap** in quite the same way as the algorithm specifies, the reasons for which have been discussed earlier (see page 78). Instead the swap is done as a single permutation of `A`, since this does not require remote bindings to be updated. The sites wishing to swap values create targets pointing to each other: to do this the width of the matrix is needed to calculate the index of the remote site. The remaining processors simply create a target pointing to themselves. **Normalise** is straight forward and here has been incorporated with **swap**:

```
(defun swap-and-norm (A matrix-width i max-row)
  (let ((swap-distance (* matrix-width (- (cdr max-row) i))))
    (get (elwise (A)
           (make-target (cond ((= (row) i) (+ (here) swap-distance))
                              ((= (row) the-row) (- (here) swap-distance))
                              (t ()))))
         (elwise (A) (if (= (row) (cdr max-row)) (/ A (car max-row)) A)))))
```

Finally **update** requires inter-processor communication where several processors will try to access the same remote processor. To avoid this we use scan to spread the row-`i` down the matrix and column-`i` across the matrix. Previously (Section 4.1.1) we were able to spread the *first* column across the matrix using $\lambda xy.y$ as the combinator. To spread a *specific* column across the matrix we must use a slightly different technique. We create a field where the only non-zero elements are the values in the column to be spread, a right prefix-sum on this field then has the desired effect.

```
(defun update (A i)
  (let ((row-i (scan (elwise (A) (if (= (row) i) A 0)) + up))
        (i-col (scan (elwise (A) (if (= (col) i) A 0)) + left)))
    (elwise (A row-i col-i)
      (if (not (and (>= (col) i) (> (row) i)))
        A
        (- A (* i-col row-i))))))   ;A[row][col] - A[i][col] * A[row][i]

(defun g-elim (A n)
  (let ((i 0))
    (while (< i (- n 1))
      (setq A (update (swap-and-norm (A n i (max-row A i)))))
      (setq i (+ i 1)))
    A))
```

4.2.1 Elementwise Parallel Prefix

In the Gaussian elimination example we saw again the utility of prefix operations for performing computations and moving data around collections of processors. We saw how TACOΣ makes it easy to use prefix operations for various arrangements of processors, and also how it is useful to be able to associate information with the paralation sites by storing data in TACOΣ object slots instead of targets. What was also noticeable in these examples is we would like to be able to create targets based on an index other than the default `index` field, for example via two-dimensional coordinates. This is not a serious limitation as the desired references can still be constructed without difficulty but it does suggest further levels of abstraction that may be appropriate and useful.

We also see in this example that although the scan operations are useful, because they are viewed as global data permutations they break down the object-oriented nature of the code. We have already encountered this problem because the paralation model separates communication and computation. This meant that when writing code for a single active object the communication would have to be *factored out* and given as additional parameters to the code. This is at odds with an object-oriented programming style and to remedy it we permitted communication within the body of an `elwise` expression. So perhaps we can allow scan operations to be invoked within an `elwise` expression in the same way.

Certainly there seems to be no functional difference between a processor participating in inter-

processor communication and it participating in a scan operation. There is however one major difficulty, and that is our current implementation of `scan` requires a global reduction to determine if there is still work to be done. So to be able to supply `scan` as an element-wise operation we need some method of allowing an individual processor to determine if there is still work remaining which may require its participation. There are various ways we can do this, for example if we know how many processors are involved in the operation then we know how many iterations are required, however this information may not always be easy to find. Similarly, if we know which processor will be last to finish then its status can be polled by other processors to see if it has completed, but again this information may not always be available.

The problem with these solutions is they require knowledge of the number and arrangement of the processors which may not be readily available, or at least tedious to keep track of. There are some other options that are rather more general:

1. Permit inter-processor writes (see original algorithm, section 4.1).
2. Make `scan` itself a primitive.
3. Supply some new primitive that makes it possible to implement `scan`.

Of these, the third option appears the most attractive as it provides a general mechanism for improving the micro-macro equivalence of the language and hence its object-oriented nature. In contrast, supplying inter-processor writes is a solution that depends on the nature of the prefix algorithm. In addition, we have been hoping to avoid the need for inter-processor writes as the problem of collisions complicates their implementation. This is also preferable to making `scan` a primitive as it compromises the orthogonality of the language by supplying an operator we cannot implement in the language.

The reduction operators of TUPLE perform a reduction for all active processors and then broadcast the result to all active processors, so the function `some-pe` can be used to determine if all processors have completed. This meets our requirements but is itself a reduction operator which we are trying to avoid implementing as a primitive.

TUPLE supplies a special conditional form `exif` which also supplies the kind of functionality we are interested in, if the consequent is executed by any of the PEs then the remaining PEs simply return `nil`. This can be used to implement an `any` function, which returns `t` if its argument is non-`nil` on any processor.

```
(exif bool () t)          ;⇒ t if bool is nil everywhere
                          ;⇒ () if bool is non-nil anywhere
```

```
(depefun any (bool)          ;So
  (not (exif bool () t)))    ;⇒ t if bool is non-nil anywhere
                             ;⇒ () if bool is nil everywhere
```

We can implement any in straight Paralation Lisp. To do this we update a singular binding captured by an `elwise` expression. In the expression below the `then` form will be evaluated on all the sites if `bool` evaluates to non-`nil` anywhere.

```
(let ((captured-singular-variable ()))
  (elwise (bool)
    (when bool (setq captured-singular-variable t))
    (if captured-singular-variable (then) (else))))
```

Though this demonstrates we can perform the kind of operation we are interested in it is quite difficult to abstract. Firstly we must assume our interpreter will capture a singular variable correctly within a function which is executed in parallel. This is non-trivial and some systems simply broadcast such values which would not give the correct result. More importantly when using nested `elwise` expressions we must capture a separate singular binding for each `elwise` form. If not interference would occur between the paralations.

We can do this by modifying `elwise` so that it captures a special singular variable when ever it executes. So although it requires some juggling it does not seem unreasonable to introduce the function any though it may not be implemented in the way presented here. We redefine `elwise` to be a macro that defines the binding `*sink*`, so if used within an `elwise` statement, a new binding will be created for it on each site executing the `elwise`.

```
(setq old-elwise elwise)


(defmacro elwise (arg-list . body-form)
  `(let ((*sink* ()))
     (old-elwise ,arg-list ,body-form)))


(defmacro any (bool)
  `(progn (setq *sink* ())                ;cancel any previous use of any
          (when ,bool (setq *sink* t))
          *sink*))
```

This construct requires synchronisation in the same way that communication within the body of an elwise statement does. As before the underlying implementation may have to force a synchronisation on some architectures but on a SIMD architecture this should not be a problem. Now that we have any available to us we can implement scan so that rather than being applied to entire fields, it is applied to their elements using an elwise expression.

```
(defun scan (data comb buddy)
  (let ((real-buddy (buddy)))
    (unwind-protect
     (while (any (buddy))
       (let ((buddys-data (get buddy data)))
         (when (buddy)
           (progn ((setter buddy) (get buddy (buddy)))
                  (setq data (comb data buddys-data))))))
     ((setter buddy) real-buddy))
    data))
```

The ability to use scans within elwise expressions is used in the next example. The code is based on matrices again but the algorithms and implementation are best expressed and understood in terms of the individual elements. We can now write code almost entirely oriented around the individual objects which make up a computation without having to break up the code for communication operations.

4.3 Artificial Neural Networks

Artificial Neural Networks have recently become a very popular method for attempting to solve a variety of problems which have inexact solutions. They attempt to model the basic organisational features of biological nervous systems. Typically they consist of a large number of simple interconnected processing elements, which model a collection of neurons and the synapses between them.

Figure 4-3 shows the basic structure of a single processing element, i.e. a neuron, in an artificial neural network (ANN). The neuron receives a set of inputs $x_0, \ldots, x_{n-1}$ through weighted links, the weighted inputs are summed and the result is passed though an output function f. The "knowledge" or functionality of the ANN is encoded in the values of its weights and various algorithms have been devised which modify these weights so that the desired input/output behaviour for the network can

be achieved.

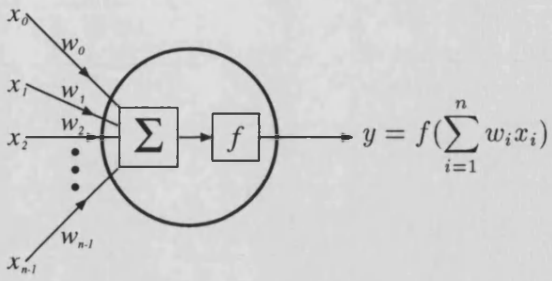


Figure 4-3: Typical "neuron" or processing unit in an artificial neural network.

Typically the cells of the network are arranged in layers (Figure 4-4), the first layer receives a set of inputs, the outputs from that layer are then fed as inputs to the next layer, until a set of outputs is generated by the final layer which represents the output of the network. The intermediate layers of the network are usually referred to as *hidden* layers.

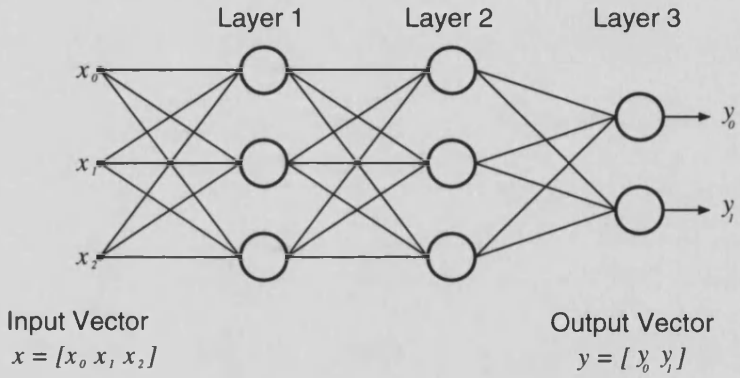


Figure 4-4: Multi-layered Artificial Neural Network

In *supervised* ANN models, a desired mapping can be found by presenting the ANN with training samples, that is providing both the input vector and the desired output vector. The ANN then computes the error between the actual and desired outputs and modifies its weights to reduce this error. Self-organising ANN models are considered to be unsupervised as no training samples are provided, the network is simply presented with the test input vectors. The hope is that the network will be able to distinguish groups of similar inputs, rather than producing an *answer* for a given input.

We will now look at a widely used artificial neural network model and outline its implementation using TACOΣ. The description here is based on that given by [35].

4.3.1 Perceptron Back-Propagation Networks

The nodes in these networks follow the basic structure given above. They also have an additional bias term θ which serves as a threshold, which can be implemented by an additional constant input 1 with its corresponding weight, w_n set to θ. The output function for each node is a sigmoid function:

$$y = \frac{1}{1 + e^{-net}} \qquad net = \theta + \sum_{i=0}^{n-1} w_i x_i$$

The back-propagation algorithm requires the output function to be differentiable. It is not really appropriate to go into these details of the algorithm's operation here, but this is a requirement for it to work.

Back-Propagation Learning Algorithm

The back-propagation algorithm is used to modify the weights in a neural network which has hidden layers. The network accepts an input vector $x = [x_0, x_1, \ldots, x_{n-1}]$ and generates output vector $y = [y_0, y_1, \ldots, y_{m-1}]$.

1. Initialise the weights and bias terms to small random values (e.g. -0.5 to 0.5).

2. Calculate the actual output vector y by propagating some input vector x from the training set forward through each layer.

3. Start at the output layer and calculate the error $\delta_i = y_i(1 - y_i)(d_i - y_i)$, where for unit i (the ith component of the output vector) δ_i is the error term, y_i is the output and d_i is the desired output. This calculation of δ_i relates the actual error magnitude, the $(d_i - y_i)$ term, to the derivative of the sigmoid output function.

4. Adjust each of the weights of the output units in proportion to the error and the input signal coming over that line. That is each connection determines how much of the error it is responsible for and modifies its weight accordingly. The change for weight w_i is given by $\Delta w_i = \alpha \delta_i x_i$ where α is a learning rate term (usually in the range $0.1 < \alpha < 1.0$) which controls the stability and speed of convergence of the network.

5. After completing the weight changes for the output, work backwards layer by layer to the first hidden layer. For each hidden layer:

   (a) The error term δ_i for the ith unit in a hidden layer is calculated as:

$$\delta_i = x_i(1 - x_i) \sum_j \delta_j w_{ij}$$

Where x_i is the units output, the δ_j are the error terms for the next layer and w_{ij} is the weight of the connection between unit i in the current layer to unit j in the next layer. Thus the error for the unit is determined from the amount of error it contributed to the next layer and its current output.

(b) The weights can now be adjusted in the same way as step 4.

6. Repeat for another input, output vector pair x, y.

At first sight this algorithm may seem rather complicated but its implementation proves to be fairly simple. It is also well suited to massively parallel architectures and can be coded to give highly parallel execution. We will first consider the process of constructing a network and propagating an input vector through it.

Let us consider a layer of the network containing m units. If the previous layer (which may in fact be the input vector) has n units then each node will have n inputs plus a bias term. The connectivity between the two layers can be represented by an $(n+1) \times m$ matrix W of the weights on each connection. Each w_{ij} is the weight of the connection between unit i in the previous layer and unit j in the current layer, row n represents the bias terms. The *net* (see Figure 4-3) for the current layer from input x is given by $net = x'W$, where x' is x augmented with an extra element with value 1.0 for the bias terms.

$$\overset{x'}{\begin{bmatrix} x_0 & x_1 & \cdots & x_{n-1} & 1 \end{bmatrix}} \overset{W}{\begin{bmatrix} w_{00} & w_{01} & \cdots & w_{0m-1} \\ w_{10} & w_{11} & \cdots & w_{1m-1} \\ \vdots & \vdots & & \vdots \\ w_{n-10} & w_{n-11} & \cdots & w_{n-1m-1} \\ \theta_0 & \theta_1 & \cdots & \theta_{m-1} \end{bmatrix}} = \overset{net}{\begin{bmatrix} net_0 \\ net_1 \\ \vdots \\ net_{m-1} \end{bmatrix}}$$

The sigmoid output function can then be applied to the elements of *net* to produce the output vector y for the current layer.

We have already looked at matrix operations using TACOΣ so much of the implementation is straightforward. We use a matrix style active class which is suitable for the neural networks.

```
(defactive-class ann-layer ()
  ((row reader   row
        initform (/ (here) width))
   (col reader   col
        initform (rem (here) width))
```

```
(up    reader   up
       initform (make-target (- (here) width)))
(right reader   right
       initform (if (= (+ (rem (here) width) 1) width) ()
                  (make-target (+ (here) 1))))
(input accessor input
       initform (if (make-target (+ (here) width)) 1.0
                  (make-target ())))
(delta accessor delta
       initform (make-target ())))
```

The `input` slot points to a site which has each links input associated with it. If we assume this value has been previously stored on the site using `update` then the function `test-layer` calculates the output for the layer, the output vector is located in the bottom row of the resulting field (as the prefix sum runs down the matrix columns).

```
(defun test-layer (weight)
  (let* ((sum-xw (scan (* weight (ref (input))) + up))
         (result (/ 1 (+ 1 (exp (* -1 sum-xw))))))
    (update result)
    result))
```

We can see that the actual calculations for the neural networks are very simple to implement. What is more interesting is the construction of the neural network. The network can be represented by a list of matrices which represent the connections between successive layers. With the exception of the first layer, the input vector is given by the output of the previous layer, and we know that this is found in the bottom row of the field produced by `test-layer`, so ideally we would like the `input` slots for each matrix to point to the appropriate site in the previous matrix. So far we have not looked at inter-paralation targets, but these can be constructed by using `mappings`. Below the function `make-layer` çreates a matrix paralation representing the connections between the previous layer and a new layer, and also creates the `input` connections to the previous layer.

```
(defun make-layer (prev-layer inputs outputs)
  (let ((new-layer (make-paralation (* (+ inputs 1) outputs)
                                    ann-layer 'width outputs)))
    (if (not prev-layer) ()
      (let* ((glue (match (elwise (new-layer) (row))
                          (elwise (prev-layer)
                            (if (>= (here) (- (length prev-layer) inputs))
```

```
                        (col) ())))))
            (targets (move (elwise (prev-layer) (make-target ()))
                           glue-map () ())))
      (elwise (targets)
        (if (targetp (input))) ()      ;;bias row
          ((setter input) targets))
        (- (random 1.0) 0.5))))))
```

The mapping `glue` matches the bottom row of the matrix `prev-layer` to the first `inputs` elements of each column in `new-layer`. Targets for the bottom row are then moved into the `new-layer` and stored in the `input` slots. The bottom row of `new-layer` represents the bias terms and their input is always 1.0, which is defined in the active class specification. We can now create a neural network given the size of each layer.

```
(defun define-ann config
  (labels ((loop (prev-layer inputs outputs config)
             (if (null-config) ()
               (let ((new (make-layer prev-layer inputs outputs)))
                 (cons new (loop new out (car config) (cdr config)))))))))
    (loop () 0 (car config) (append (cdr config) '(1)))))
```

Although the construction of the network is quite verbose in places it is well worth the effort since it greatly simplifies the code to process the network. The result of `define-ann` is a list of fields, each field represents the weights connecting two successive layers and the paralations have been appropriately connected together. The function `test-ann` generates an output field for a given input vector.

```
(defun test-ann (ann input-vector)
  (labels ((loop (layer-list)
             (let ((layer (car layer-list)))
               (if (null (cdr layer-list))
                 (elwise (layer) (ref (input)))
                 (progn (elwise (layer) (test-layer layer))
                        (loop (cdr layer-list)))))))))
    (let ((first-layer (car ann)))
      (elwise (first-layer) (update (vector-ref input-vector (col))))
      (loop ann))))
```

The first layer is a special case and has the input vector explicitly stored where it can be reached. Thereafter each matrix reads its input and calculates the output which it makes available for the

succeeding matrix. The final iteration extracts the ann output into a paralation of the correct size.

We now understand all the mechanisms we need to implement `train-ann`. The propagation phase is the same as in `test-ann`. Having found the output we create new weight fields based on the error in the succeeding layer. For this, each unit requires a `delta` slot connecting it to the the appropriate cell in the next paralation. The `delta` connection is the reverse of the `output` with each row being connected to the first column in the next matrix, we will omit the details and assume `define-ann` defines `delta` correctly. The function `correct-layer` calculates new weights for a layer given the layer input and current weights assuming the next layer has already been corrected and calculated the δ for this layer.

```
(defun correct-layer (input weight)
  (let* ((error (ref (delta)))
         (sum-dxw (scan (* error weight) + right)))
    (update (* sum-dxw (* input (- 1 input))))      ;δr = xr(1 − xr) Σj δj wij
    (+ weight (* error input))))                    ;Δwi = αδi xi
```

This function is fairly straight forward but an important part is the way it calculates the δ terms for the previous layer. This is similar to the way `test-layer` calculates the output which is used as input by the next layer.

```
(defun train-ann (ann input-vector output-vector)
  (labels ((loop (layer-list)
             (if (cdr layer-list)
               (let* ((weight (car layer-list))
                      (input (elwise (weight) (ref (input)))))
                      (dummy (elwise (weight) (test-ann weight)))
                      (result (loop (cdr layer-list))))
                 (cons (elwise (weight input)
                         (correct-layer input weight)) result))
               (cons (elwise ((last (car layer-list)))
                       (let ((result (ref (output))))
                         (update (* (- (vector-ref output-vec (here)) result)
                                    (- 1 result) result))    ;δi = yi(1 − yi)(di − yi)
                         result)) ()))))
    (elwise ((input (car (ann))))
      (update (vector-ref input-vector (row))))
    (loop ann)))
```

4.4 Connectionist Networks

Connectionist networks are another class of neural network and they have much in common with the artificial neural networks described in the previous section. As with the ANN a large number of computing elements are connected by weighted links, but connectionist networks operate rather differently to ANNs. Each element of the network has an *activation level*: an *input* to the network is some initial set of activation levels, i.e. an initial state for the network. The network then computes a corresponding output state. To do this the units update their activation levels so that they harmonise more closely with the weighted sum of their neighbour's activation levels. There are similarities between the algorithm for modifying the activation levels and the back-propagation algorithm. The levels are repeatedly modified either until the network stabilises or for a fixed number of iterations. Another feature of the connectionist networks is the structure of nodes and connections actually matches that of the problem. This is why a *start state* represents meaningful input to the network. This is quite different from the artificial neural networks where a general structure is heuristically tuned to give a structure with the desired properties.

We give here a description of connectionist networks based on their use in a knowledge representation and inference system [22]. This system consists of a declaration language called NEULA (NEUral LAnguage) which compiles collections of object descriptions into a connectionist network. The language supplies various mechanism for interrogating the knowledge base that the network represents, these enquiries are converted into an input state for the network which is then executed and the completion state is interpreted by NEULA to give a response. Thus NEULA is a high-level knowledge representation language which interfaces with a connectionist network package (The Rochester Connectionist Simulator [23]) to perform inference operations.

Below is a typical NEULA object description for an object *Hobbit*. This binds *Hobbit* to a set of triples (P, V, W), where P is an attribute, V is the attribute value and $C \in [0.0, 1.0]$ is the confidence factor. The confidence can be specified by a key word which has a value associated with it, e.g. *many*, the confidence defaults to *all*.

OBJECT Hobbit is Middle_Earth_Inhabitant

~nature	good
~is_fond_of	round_things
~**not** is_fond_of	swimming(many)
~life	mortal

NEULA also supplies various shorthands and mechanisms for specifying other relationships

between objects, some examples are, **not** which specifies a negative relationship, mutual exclusion, symmetry and nonreflexivity.

The process of converting the object descriptions into a connectionist network proceeds in a relatively intuitive[1] fashion.

1. A *label* unit is created in the network for each object.
2. An *attribute* unit is created for each triple $(P_{i'}, V_{i'}, W_i)$.
3. An arc is drawn between the label unit and each of its attribute units. The weight of this arc is given by the certainty (W_i) of the attribute triple.
4. Each label unit is connected by an *is-a* arc to the label unit the object inherits from.
5. An *echo* arc is added for every existing genuine arc, this permits an object to be recognised from its attributes. The arc runs in the opposite direction and its weight is some fraction of the genuine arc's weight.
6. If a label and a label (either directly or indirectly) it inherits from have attributes which are mutually exclusive, (e.g. Hobbit's nature is good but Gollum is a hobbit with an evil nature) then correcting, negative, arcs are added (so Gollum has a negative effect on the good nature of hobbits and vice-versa).

We can construct a NEULA style connectionist network using TACOΣ in much the same way we would build an ordinary data structure to represent the network. The network has two distinct components, units and arcs, we define active classes for each of these.

```
(defactive-class c-net-component ()
  ((input accessor input)))        ;Both arcs and units have one physical input

(defactive-class unit (c-net-component)
  ((name reader name               ;The unit name within the network
         initarg name)
   (type reader type               ;i.e. label, is-a etc.
         initarg type))
  constructor (new-unit type name)
  predicate is-unit)
```

[1] In this system the *Bilbo* in *(bearer, Bilbo)* is different to the Bilbo that *(is-a, Hobbit)*, this aspect seems less intuitive.

The unit name is either an object label, or a key constructed from the relation and value. These names are unique within the network and we use a table to ensure this:

```
(deflocal unit-table (make-table))

(defun make-unit (type name)
  (let* ((name (if (eq type 'label) name
                   (make-symbol (format () "~a:~a" type name))))
         (exists (table-ref unit-table name)))
    (if exists exists
      (let ((new (new-unit type name)))
        ((setter table-ref) unit-table name new) new))))

(defun label (name) (make-unit 'label name))
```

For the purposes of this example we will only support the most primitive mechanisms supplied by NEULA. We are interested here in building an active connectionist network which can take advantage of the inherent parallelism. We are not concerned with the separate problem of defining a connectionist network from a collection of object specifications. In view of this, an object specification will merely be a list of relations and values, i.e. there will be no support for **not**, mutual exclusion etc.

```
(defun attribute (attr-list)     ;(relation value ...)
  (make-unit (car attr-list)
             (cadr attr-list)))
```

Arcs connect the units in the network and there will, in general, be several arcs entering any unit. We define an active class for arcs which has an input from a single unit and a weight. The arc also has a *next-arc* slot which can point to another arc which is going to the same unit. So effectively we model several input arcs to a unit as a linked list of the arcs, with the head of the list connected to the unit.

```
(defactive-class arc (c-net-component)
  ((weight
    initarg weight
    reader weight)
   (next-arc
    initarg next-arc
    reader next-arc))
  constructor (make-arc input weight next-arc)
  predicate is-arc)
```

If we create an arc *from* one unit *to* another, then the *to* unit receives input from the *from* unit. So the input of the *to* unit is set to an arc that reads its input from the *from* unit. Notice that the previous input to the *to* unit is preserved in the `next-arc` slot.

```
(defun arc (from to weight)
  ((setter input) to
   (make-arc from weight (input to))))
```

Given a label unit and a list specifying its attributes we create corresponding attribute units and connecting arcs.

```
(defun add-attributes (to attr-list)
  (if (null attr-list) to
    (let ((attr-unit (attribute attr-list)))
      (arc to attr-unit 1.0)                ;label strongly activates attribute
      (arc attr-unit to .33)                ;attribute weakly activates label
      (add-attributes to (cddr attr-list)))))
```

If an object's declaration specifies an *is-a* object, the object inherits from this object. This means that if a label is activated then both its own and its inherited attributes should become activated. Arcs are created between the label units accordingly.

```
(defun inherits (unit name)
  (if (null name) ()
    (let ((is-a-unit (label name)))
      (arc unit is-a-unit 1.0)
      (arc is-a-unit unit .33))))
```

This gives us all the machinary we need to specify an object using syntax similar to that used by NEULA:

```
(defmacro OBJECT (name is-a is-a-name . attribute-list)
  `(let ((new (label ',name)))
     (add-attributes new ',attribute-list)
     (inherits new ',is-a-name)))

;; Usage example: (OBJECT Hobbit is Middle-Earth-Inhabitant
;;                        nature good
;;                        height short)
```

Once all the object declarations have been read, the units and arcs that have been created must be collected into a single paralation so that we can execute code on the network in parallel. If we assume

the network is connected then this is straightforward: we can simply choose a unit at random and generate the desired paralation from it using `connected`. For a disconnected network the enquiry will effectively specify which units to generate a connected network for.

```
(setq c-net (connected (table-ref unit-table (car (table-keys unit-table)))))
```

We now describe the *iterative activation propagation method* which generates inferences from the network. The units are given a starting activity level, such that those units of interest have an activity of 1.0 and all other units an activity of 0.0. All the units then change their activity level according to the *activation propagation formula.* This process is repeated for either a fixed number of iterations or until the network stabilises.

If a single node with activity a is connected to nodes a_i, where $i = 1, n$, by arcs with weights w_i, then the new activity for the unit, a', is given by the formula:

$$a' = a + \delta(a) sigma(\sum_{i=1}^{n} w_i a_i)$$

Much of this is similar to the computations performed in artificial neural networks. We have a sum of weighted inputs which is passed through an output function *sigma*, this is defined as:

```
(defun sigma (x)
  (- (/ 2 (+ 1 (exp (* -1 x)))) 1))
```

$$sigma(x) = \frac{2}{1 + e^{-x}} - 1$$

The amount to modify the weight by is then calculated. This is related to the output and the current activity level. A suitable definition of δ is:

```
(defun delta (x) (- 1 (abs x)))
```

$$\delta(x) = 1 - |x|$$

As before, we consider the operations performed by individual sites depending on their active class. The activation propagation formula falls naturally into three stages:

```
(defun do-arc (activity)                          ;read and weight input
  (let ((input (get input activity)))
    (if (is-arc) (* input (weight)) 0.0)))

(defun sum-arcs (value)                           ;scan-add weighted inputs
  (scan value + (if (is-arc) next-arc (lambda () ()))))   ;over next-arc links

(defun do-unit (activity psum-value)              ;read weighted sum and
  (let ((input (get input psum-value)))           ;calculate new activty
    (if (not (is-unit)) 0.0
      (* (delta activity) (sigma input)))))
```

The state of a network can be represented by a field in the networks paralation. The field elements will be floating point numbers, on `unit` sites the value will be the unit's activity level and on `arc` sites it will simply be 0.0. We can now write a function which given an input state will run the activation propagation model for a given number of iterations and produce the corresponding output state.

```
(defun run (activity iterations)
  (if (= iterations 0) activity
    (run (elwise (activity)
           (do-unit activity (sum-arcs (do-arc activity))))
         (- iterations 1))))
```

Connectionist networks prove to be an excellent example for TACOΣ. They are naturally suited to an object-oriented implementation: here we saw that much of the code (particularly the construction phase) closely matches the steps in the algorithm. In addition, while being an irregular and heterogenous network it is still able to make use of the powerful `scan` operator. This is particularly useful in this example since `scan` effectively induces a binary tree on the arcs, we could have built such a tree explicitly but it would have been rather more complicated. That linked lists are often just as good as binary trees in data-parallel execution is fairly unintuitive [62] and can be very important when programming with TACOΣ.

This implementation has the advantage that the network can be modified and then further runs be performed. This highlights the advantages of TACOΣ over straight paralation lisp for applications of this nature. To add a node requires allocating a new paralation, moving all the data and re-generating the mappings. To simply change a connection will require regenerating the mappings. For TACOΣ we merely allocate one new site and change some slot values. Further to build a paralation and set of mappings that represent a network, all the connections must be determined beforehand, i.e. it cannot be done incrementally. This process would probably be much the same as the network construction code given here. Whereas the final structure would then have to be converted to a paralation and some mappings, the TACOΣ data structure is already able to execute code. In this way TACOΣ can simplify the task of organising a problem so that it is suitable for data-parallel computation.

4.5 The Paralation Lisp Function Library

As we discussed in section 2.4, Paralation Lisp supplies a library of powerful high-level functions. Although these functions can be written using Paralation Lisp [55, pages 113-118], the implementations are quite complex. This is partly the motivation for TACOΣ, since if the implementation of such useful functions proves difficult then perhaps the kernel of Paralation Lisp is inadequate. In this

section we will give TACOΣ implementations for some members of the library.

First we observe that most of these functions are performing operations on fields as though they were sequences. For example `expand` which is used by `collect` is a concatenation operation and is implemented in terms of the primitive operator `field-append-2`. This suggests that we need to make all our ordinary paralations instances of a TACOΣ sequence class.

```
(defactive-class sequence ()
  ((next
    accessor next
    initform (+ (here) 1))))


((setter default-aclass) sequence)
```

This means that each site of a simple paralation will now be an instance of the TACOΣ class `sequence`, so each site will have a `next` slot pointing to its next immediate neighbour. The function `field-append-2` takes two fields and returns a new paralation containing the two fields in sequence. Below we give an implementation of `field-append-2`. This uses `project` which we have yet to make much use of and also the function `target-of`, this is useful shorthand for `(value (elwise (p) (make-target)))` and it is also faster as is accesses information stored by TACOΣ.

```
(defun field-append-2 (field1 field2)
  (let ((new (connected (elwise (field1)
                          (when (null (next))
                            ((setter next) (target-of field-2)))))))
    (elwise ((field-1 (project field-1 new))
             (field-2 (project field-2 new)))
      (if (eq field-1 '*nothing*) field-2 field-1))
```

This does not have quite the same functionality of `field-append-2` because the result will not have any site repetitions. So if we try to `field-append-2` fields from the same paralation we will simply get another field in the same paralation. This is not necessarily a problem, since although it is not quite the same as `field-append-2` of Paralation Lisp it is a perfectly reasonable operator in the context of TACOΣ and one wonders why we should be trying to append a paralation to itself in this way. Rather than implementing `expand` in terms of `field-append-2` it is easier to implement it directly:

```
(defun expand (fields)
  (elwise ((field fields))                          ;for each of the nested fields
    (let ((next-one (get next (target-of field))))  ;get the target of the next field
```

```
    (elwise (field)                                          ;and then append each
      (when (null (next)) ((setter next) next-one)))))       ;field to the next
(let ((new (connected (field-ref fields 0))))                ;collect field paralations
  (elwise ((field fields))                                   ;into single paralation
    (elwise (field) (update field)))                         ;and move fields into
  (elwise (new) (ref (make-target ()))))                     ;new paralation
```

Here we use `update` and `ref` to move the fields into the larger paralation. This is preferable to `projecting` each field in turn and then merging the resulting fields. The use of `expand` is discussed further in section 5.3.3 with respect to its implementation and use in a quicksort function.

Another advantage of making our paralations default to the `sequence` active class is there will always be a hook for prefix operations over the paralations, and they can be performed at any level of nesting. We have repeatedly seen in this chapter the utility of the `scan` operation so this is important. Further the prefix operation can be performed regardless of the physical arrangement of the processors and indeed for sets that inter-leave with each other. This means we can supply much of the functionality of a language like NESL while imposing fewer restrictions. In Chapter 6 we look at the implementation of communication in TACOΣ and NESL, and see that the complexity of many operations is the same for both TACOΣ and NESL but while NESL must adhere to a regime of contiguous, segmented collections of processors, in TACOΣ the location of processors is unimportant. Thus the usefulness of the constructive paradigm is not made available at the cost of other important mechanisms.

Chapter 5

Issues in Implementation

In this section we will discuss some of the key issues in the implementation of functional data parallel languages, with particular reference to the various mechanisms that are required by TACOΣ. We will start by describing a fully operational implementation of the Paralation EULISP interpreter developed at Bath. This system was developed with the requirements of TACOΣ specifically in mind. As such it will give us a good basis for identifying the kernel operations of functional data parallel languages in general as well as those needed by TACOΣ and then discussing their implementation.

5.1 BLINDPEU

BLINDPEU[1] is the name given to the bytecode interpreter developed for the *MASPAR* MP-1 at Bath on which the implementation of Paralation EULISP is based. As described earlier in section 2.1.2 the *MASPAR* is a self-contained, subsystem capable of executing both parallel and serial code which is connected to a conventional host computer (see also Appendix A). Although the *MASPAR* is able to execute serial programs it is not suitable for running a full lisp system, since the processor is not very powerful and more importantly has very little local memory (128k). The natural thing to do is run the lisp system on the host computer which will make calls for code to be executed on the *MASPAR* as needed. An earlier version of Paralation EULISP based on Plural EULISP (*c.f.* Section 2.1.3) proved too slow to be practical. This is because the host controlled the execution of the *MASPAR*, making a great many call requests to primitive functions, in much the same way the Connection Machine is controlled by its host. On the *MASPAR* however the overhead of communication with the host is too high for this to be viable. For this reason it was necessary for the *MASPAR* to execute lisp expressions independently. Some other factors also affecting the design included:

[1] **B**ytecoded **L**isp **I**nterpreter for **D**ata **P**arallel **EU**LISP

1. Reasonable execution time
2. Easily extended to support new language constructs
3. Interest in MIMD emulation by SIMD computers
4. Compact representation of functions (limited data space on *MasPar*)
5. Support for virtual processors

A data parallel byte code interpreter for the *MasPar* seemed to fit these requirements well. To execute a lisp expression on the *MasPar* it would have to be compiled into a bytecode vector. This could then be transferred to the *MasPar* in a single operation where it would then be interpreted. In addition, if the bytecode interpreter were capable of MIMD emulation it would give the *MasPar* the appearance of a multi-computer, so the work could also be applicable to these architectures. Briefly BlindPeu's features include:

Parallel Lisp Interpreter: Lisp expressions executed in parallel on each element of the processor array.

Virtual Processor Mechanism: Each physical processing element emulates several virtual processing/communication sites.

Virtual Interpreter Mechanism: If a physical processor has to execute code for several virtual processors simultaneously they are run in pseudo-parallel by inter-leaving the instruction streams.

Support for Nested Parallelism: Nested `elwise` expressions are executed fully in parallel at all levels.

MIMD Emulation: Each virtual interpreter has completely independent state allowing different code streams to be executed by different interpreters.

We now outline the implementation of BlindPeu, giving details of the memory organisation, the operation of the interpreter and its interaction with the controlling EuLisp process.

5.1.1 Memory Organisation

Figure 5-1 outlines how the memory of a single processing element is utilised. Each PE has an array of register sets giving the state of a fixed number of virtual interpreters. All bytecode vectors are

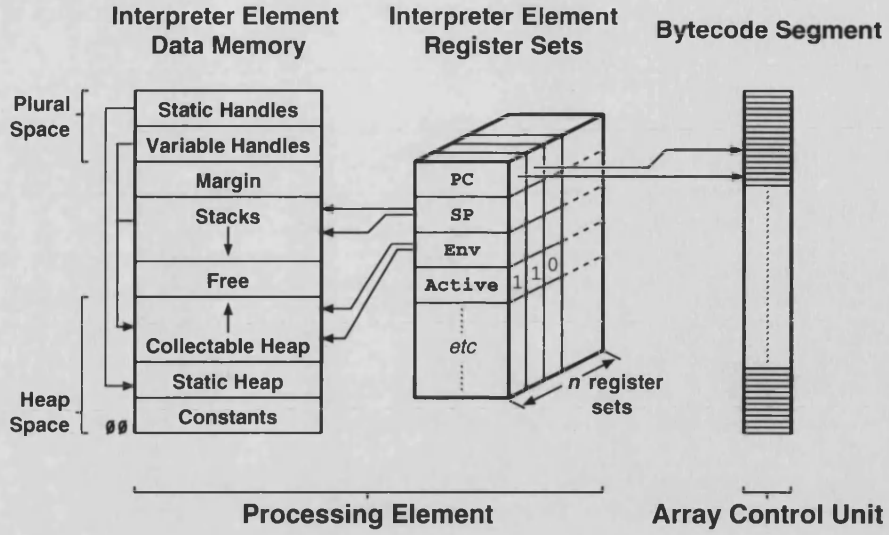


Figure 5-1: PE Memory Organisation

stored in a memory segment on the ACU. The program counter for each virtual interpreter is a pointer into this code segment. Although it is more expensive to dereference these inter-processor pointers, having all the code in one sharable location makes more data memory available on each PE. The virtual processors and interpreters emulated by each PE share a single memory segment which is used for the heap, stacks and various other tasks. Because of the limited space on each PE the memory segment is treated as a 16-bit address space. The rest of this section explains the significance of each part of this memory segment.

The Plural Space

This section of the memory is used to give handles on collections of objects allocated on the processor array. In order to identify an object on each of the processing elements a slice of the plural space, that is the same memory location on each PE, is allocated and the address of the object is stored in this location. In this way only a single value is needed to specify the entire collection.

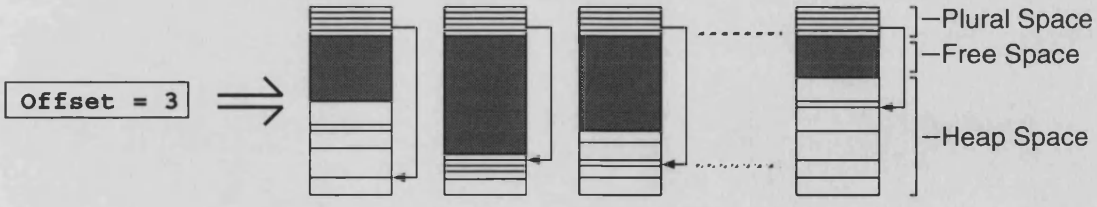


Figure 5-2: Single Integer Specifying Parallel Collection

In order to conserve plural space, a slice of the plural space can be shared between disjoint collections of PEs. Thus, to specify a collection of objects we need, in addition to the plural space

slot, to know which PEs belong to the collection. To do this we use another plural space slice of `nil` and non-`nil` values which we call the *context*, on those PEs belonging to the collection, the value in this plural space slice will be non-`nil`. In this way a collection of objects can be specified by two plural space offsets, one for the context and one for the actual objects.

The plural space is located at the high end of the memory segment and grows downwards as needed. For the purposes of garbage collection the plural space is divided into two regions:

Static Handles hold objects associated with compiled lisp functions and cannot be garbage collected. The plural space location is stored in the bytecode vector by the linker.

Variable Handles are used to specify parallel variables, i.e. fields. It is clear that a paralation can be specified by a context offset and a field by a plural space slice allocated in that context. These plural space locations can be collected and reused.

To supply virtual processors, that is give the appearance of more processors than are physically present, BLINDPEU permits each plural space slice to hold multiple values. In general, the contents of a plural space slice will be a vector containing one or more objects, these are referred to as *overloaded* values. Thus to represent a field in a paralation with 2000 elements on a 1024 processor machine, 976 PEs would have 2-element vectors and the remaining 48 would have 1-element vectors. The interpreter runs in overloaded mode for these overloaded values, emulating as many virtual interpreters (up to a fixed limit) as are needed (*c.f.* section 5.1.2).

Margin

This is a predefined gap between the plural space and the stacks. This allows the plural space to be extended while the stack is being used. This is likely to happen if nested parallelism is being used.

The Stacks

In order for BLINDPEU to support virtual interpreters it needs multiple stacks. The stacks start a fixed distance away from the plural space and grow towards the low end of memory. Rather than pre-allocate some fixed stack space for each virtual interpreter the stacks are inter-leaved with each other. To do this the gap between each entry for a particular stack corresponds to the maximum number of virtual interpreters needed, currently this is specified by the user. This means the stacks grow faster than necessary but does not place an artificial limit on the size of the stacks.

Figure 5-3 shows a possible state of the stacks with a maximum of four virtual interpreters two of which are active. All stack operations are done relative to the variable `stackbase`, so that in the

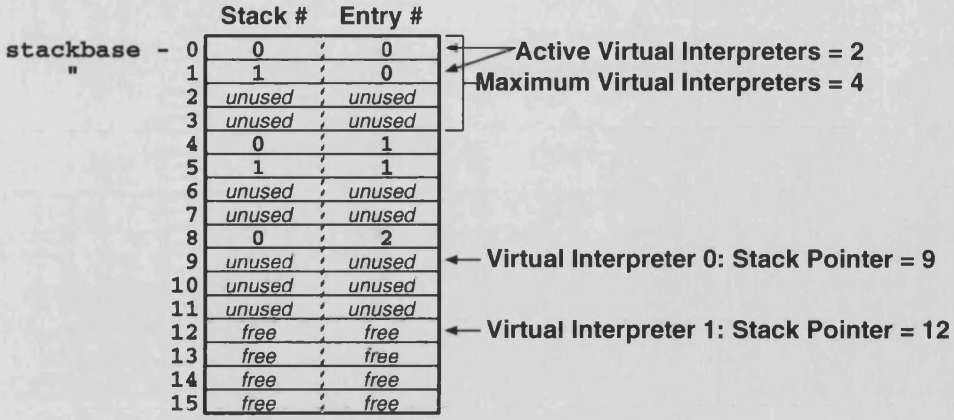


Figure 5-3: Multiple Inter-leaved stacks

event that the Margin is not big enough, the stacks can be shifted during execution to create more space.

Free Memory

All the processing element's free memory lies between the stacks and the top of the heap by virtue of a compacting garbage collector[41, 25, 56] which is invoked for all processing elements if the stacks and heap are about to clash on any of the PEs.

The Heap

The Heap contains all the allocated lisp objects. Allocation is simply done by increasing the heap top pointer. The heap is divided into three regions:

The Collectable Heap contains all objects allocated during execution, these may be reclaimed by the garbage collector.

The Static Heap contains objects pointed to by the static plural space handles and these cannot be collected.

The Constant Heap contains objects like `nil`, `t` and `*unbound*`. Naturally these cannot be collected either.

Characters and small fixnums ($\pm$8192) are immediate data, all other types have a 16-bit header. The header high bit is a GC flag, 5 bits are used for the object's type and the remaining bits give the object size in bytes. In the context of the limited memory on the *MasPar* this has not yet been found at all limiting.

5.1.2 Interpreter Operation

The interpreter is invoked with a list of plural space offsets, a bytecode function pointer and an operation mode flag. The first plural space offset gives the context for the operation, i.e. which processing elements are participating. The remaining offsets give the arguments for the function on each processor. The interpreter has two modes of operation: simple and overloaded.

For *simple* execution each processing element initialises a stack and register set. A completion frame is pushed onto the stack, the interpreter is marked as active, the program counter set to the function address, and the contents of the plural space slices are pushed onto the stack of each interpreter. The interpreters are started and each executes until it encounters the completion frame, when all the interpreters have finished the results are copied from the top of each interpreter stack into a new plural space slice.

For *overloaded* execution the contents of every non-`nil` element of the context plural slice will be an overload vector. The processing elements initialise virtual interpreters for each element in the overload vectors. The initialisation process is much the same as for simple execution, an additional validation phase is needed to check that on each PE all the overload vectors, i.e. the context and arguments, are the same size. When pushing the arguments on to the stacks the appropriate value is extracted from the overload vector and placed on the corresponding virtual interpreter's stack. The interpreter is then invoked and when all the virtual interpreters have completed the results are collected in overload vectors on each PE and these are placed in the new plural space slice.

5.1.3 System Operation

Here the general organisation (figure 5-4) of the system and its operation are outlined. A simple module mechanism is supplied which allows a EuLisp module to be compiled separately to produce a bytecode object file. When the system is started, the object files for all the modules being used are loaded into the ACU code segment by the linker (dotted lines). The linker is also responsible for allocating any lisp objects associated with functions. When all the modules have been loaded the system is in a static state, i.e. none of the components: code vectors, lisp objects, plural space handles, are collectable. The current pointers are stored for later use by the garbage collector and this gives rise to the divisions described in the previous sections.

`Elwise` is a macro which expands into a call to the function `pcall`. This is a special C-function added to EuLisp which invokes parallel execution on the *MasPar*.

(`pcall` *context-offset function-address argument-offset+ loaded*)

Where:

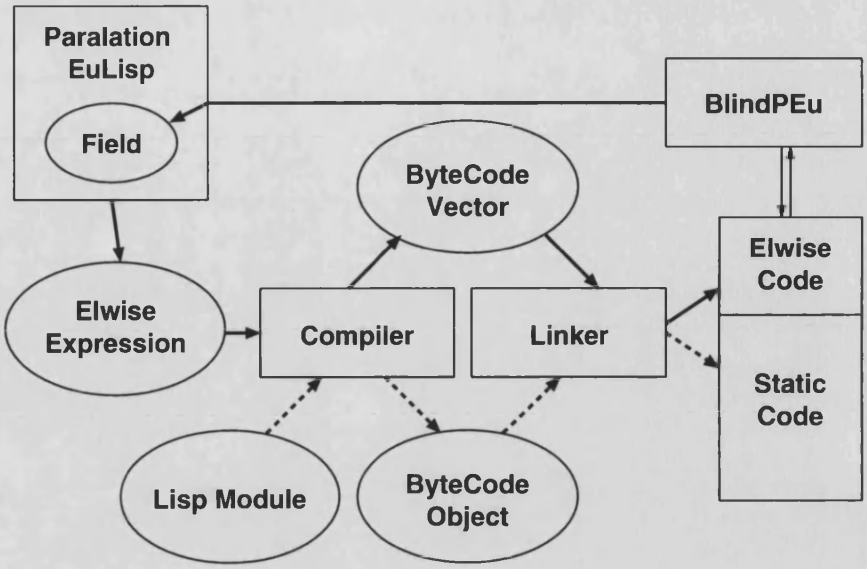


Figure 5-4: Runtime Organisation

context-offset specifies a slice of the plural space identifying the processor set being used.

function-address is the start of some function in the ACU bytecode segment.

argument-offset specifies a plural space slice which contains the values to be passed as arguments to the function.

loaded indicates whether overloaded or simple operation is required, for `elwise` this will be overloaded.

The body of the `elwise` expression is expanded into a lambda expression which is compiled, and loaded into the ACU code segment by the linker. The linker returns the vectors address which is passed to `pcall`. The various plural space offsets are all held in the EuLisp objects representing fields and paralations these offsets are extracted and passed to `pcall`. The result of `pcall` will be another plural space offset which is packaged up in a new field object using the same paralation as the parameter fields. Once the expression has been evaluated the bytecode vector can be discarded and so the next vector will be stored starting at the same location.

5.2 Supporting Virtual Processors

In the previous section we gave a fairly brief overview of BlindPeu's implementation and operation. Though BlindPeu is essentially a simple bytecode interpreter implemented on a data-parallel

architecture the techniques used to support collections larger than the physical array size are of particular interest. These mechanisms give the appearance of there being more processors than are physically present but in the description given so far there is no concept of location associated with these processors. That is to say we are simply able to process more values than there are physical processors, *where* these *new* processors actually *are* is not apparent. This is why we make the distinction between virtual processors and virtual interpreters, in other virtual processor mechanisms – like that on the Connection Machine – this distinction does not exist. In this section we will describe the virtual processor mechanism supplied by BLINDPEU and compare it to that used by the Connection Machine. We will also see how the mechanism is suitable for allocating classified paralations and TACOΣ operations in general.

5.2.1 Why Do We Need Virtual Processors?

There are some schools of thought that say we do not need virtual processors and that they are in fact a bad idea [57, 18]. However the virtual processors referred to here are indistinguishable from physical processors right down to the instruction level, and indeed the physical processors themselves must be accessed via this virtual processor mechanism. We use the term in a rather more general fashion, and view any system that gives the appearance of more processors than are physically present as a virtual processor mechanism.

The need for virtual processors in this form is of course obvious, there are still a large number of problems that will not fit neatly onto even our largest computers. We may leave the task of virtualisation to the programmer but this is time consuming and results in non-portable code. By supplying virtual processors the programmer can be insulated from the physical details of a platform improving productivity and portability. We may leave the task of virtualisation to the programmer arguing that this will lead to more efficient code, but doubtless most good programmers will soon find the need for a virtualisation library, which they would implement themselves. Supplying virtual processors within the language allows us to supply an efficient implementation based on in-depth knowledge of the architecture.

The paralation model permits a paralation of any size to be allocated, so clearly some virtual processor mechanisms will be needed. The question is how authentic should these virtual processors be? In BLINDPEU virtual processors are supported at the bytecode instruction level. This arrangement was both necessary and practical, however the mechanism used is rather different from that on the Connection Machine.

5.2.2 Virtual Processors in Paralation Lisp

As stated in the previous section the paralation model requires virtual processors as a paralation can be of any size and also we can create multiple paralations.

While creating the need for virtual processors it also greatly simplifies their requirements. This is because the paralation model strictly separates communication and computation so that side effects between sites cannot occur. This means the body of an `elwise` expression can be re-evaluated for different paralation sites without danger of interference from previous runs. Because communication is always a single monolithic operation its implementation is greatly simplified.

To illustrate this we will briefly describe how, given a primitive Paralation Lisp without a virtual processor mechanism, it is then straightforward to implement a version that can support paralations of any size. The primitive Paralation Lisp will only be able to create paralations that have less sites than the physical number of available processors. However it must be able to create as many of these *primitive* paralations as are needed. Such a system is fairly simple to implement and is used in Plural EuLisp, a full description of which can be found in [40].

We have already seen, in Section 5.1.1, how virtual processors can be supported by storing vectors of objects on each processor. We can think of this as representing a field with a primitive field of vectors. An alternative approach is to represent a field with a vector of primitive fields. A paralation of n sites, where n is greater than the physical array size N can be represented by a collection of primitive paralations, where no more than one has less than N elements and the rest have N elements. Because there are no inter-site dependencies within an `elwise` expression the body can be re-executed for each primitive paralation without danger of interference. To create mappings between paralations we must create a primitive mapping between every source–destination pair of primitive paralations. To move a field down a high-level mapping we must perform a primitive move for each primitive mapping it contains and merge the results to create the resulting high-level field. This method was used for a Plural EuLisp based implementation of Paralation Lisp.

5.2.3 Virtual Processors for Active Objects

In the previous section we saw how the Paralation Model greatly simplifies the requirements of a virtual processor mechanism. However this is not the case with TacoΣ which places much more emphasis on the identity of the paralation sites. This is because TacoΣ is oriented towards the construction of structured paralations. To understand the difference TacoΣ makes to paralation lisp consider a union of two paralations.

With Paralation Lisp the union must be represented by a completely new paralation but TacoΣ can

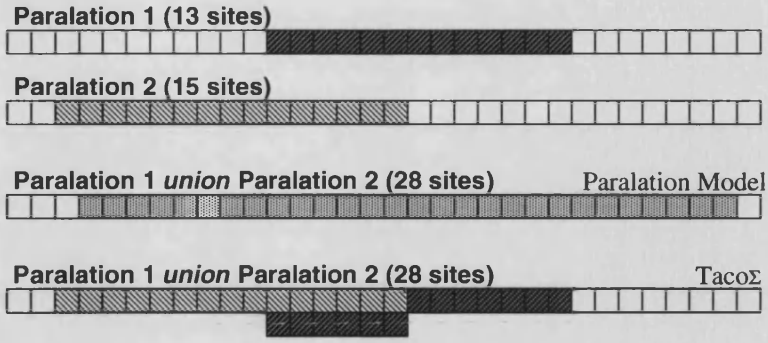


Figure 5-5: Creating a Union of two Paralations

collect all the sites of the two paralations into a new paralation. This represents a more authentic union operation since the original paralations really are contained within the new paralation. But because we are at the mercy of the processor allocator the two paralations may share physical processors, i.e. the virtual processors making up the paralations may reside on the same physical processor.

As the identities of the original paralations are preserved by the union, i.e. we know which sites in the union belonged to which of the original paralations, it is not sufficient merely to *overload* a physical processor. It is necessary for the overloaded processor to give the appearance of multiple distinct processors, i.e. the overloaded values should be associated with sites having a unique identifier or address.

5.2.4 Virtual Processors on the Connection Machine

The Connection Machine has a high level instruction set called Paris (**Par**allel **I**nstruction **S**et) implemented in microcode. Paris supplies instructions for creating and using virtual processors; the programmer can define a geometry using `create_geometry` and then allocate a virtual processor set of that geometry using `allocate_vp_set`. The geometry specifies how many dimensions the set has and the length of each axis: the lengths must all be powers of two and the product of the lengths must be a multiple of the physical number of processors on the Connection Machine that the controlling process is currently attached to. (The Connection Machine can allocate subsets of the array to separate programs, possible options are 4k, 8k, 16k, 32k and 64k.) The operators `allocate_heap_field` and `allocate_stack_field` are then used to allocate memory on the virtual processor sets, where a field is a memory segment at the same location on every processor. These operators reflect the virtual processor mechanism used by the Connection Machine where the memory of each physical processor is repeatedly halved to give separate memory segments for the virtual processors being emulated by each processor.

Paris supplies instructions to tune the layout of the virtual processor sets to make the best use

of the communication networks and as such supplies a simple interface to the Connection Machine hiding many of the problems of virtualisation. However being such a high-level instruction set is arguably a bad thing [57] and "strip-mining" techniques in compilers generating code for a RISC style instruction set would produce more efficient code. The processor allocation mechanisms supplied by *Lisp are very similar to those in Paris.

5.2.5 Virtual Processors in BLINDPEU

Much of the design of BLINDPEU is motivated by an interest in using a SIMD processor array for heterogeneous computation. Finite state automata provide a good example of how this can be done. FSA are very simple machines consisting of a transition table, a state and a sequence of inputs. The operation of the FSA is simply a series of state transitions based on the current state and input. by placing an FSA state on each PE, a single SIMD FSA program could process a different input on each PE. As a more complex example, the PEs could hold the FSA transition table as well as the state, this would allow different FSA to be run on each PE by a single, general program. BLINDPEU represents a much more extreme example of heterogeneous computation where one program can execute completely different lisp expressions on each PE. This seems a long way removed from the FSA, but most lisp programs consist chiefly of a small number of operations, function application, object reference etc. So though more complicated than an FSA the principle is the same.

These goals lead to an execution model which will tend to load processors unevenly, depending on their respective tasks. This made the Connection Machine's system of supplying virtual processors unattractive as it places an artificial limit on the virtual processors available memory which may be inappropriate. It is easy to visualise one virtual processor exhausting its memory segment while other segments on the same physical processor remain unused. For this reason a system where the virtual processors share the same physical heap on each PE seemed more sensible. In BLINDPEU the virtual processor mechanism is independent from the memory management system (apart from the fact it uses heap objects to manage virtual processors!)

Although we have argued that the Connection Machine's virtual processor mechanism is a poor system for the kind of computation we are interested in it should be pointed out that it *is* an appropriate mechanism for the Connection Machine. This is because there are some important differences in the architectures of the *MASPAR* and the Connection Machine. Firstly the local memory associated with each PE is much smaller on the Connection Machine than on the *MASPAR*, somewhat less than 4k on the CM-2 as opposed to 16 or 64K on the *MASPAR*. Secondly, and perhaps more importantly, the *MASPAR* supports local indirect addressing, this means that although each PE is executing the same

instruction stream, the instructions can be applied to data at different addresses on each PE, this is not possible on the CM-2. This means that the Connection Machine would have been an inappropriate platform for the applications we are interested in, the nature of the architecture makes it better suited to dealing with n-bit strips of the processor array (32-bit integers for example) in a uniform fashion. In the CM-2's defence we must remember it forms a single address space with its host, so these n-bit values can be pointers to objects on the host. This is not the case on the *MasPar*, but then its processors have the power and resources to allocate and manipulate such objects locally. In TUPLE only[2] cons cells are allocated in the PE memory and all other non-immediate data is stored on the host. Although this clever implementation gives the *MasPar* a global address space with its host, it does mean that operations on these objects, apart from comparison with `eq`, are very expensive.

The virtual processor mechanism in BLINDPEU can be thought of as objects maintained by a virtual processor environment, which is referred to collectively as a *Virtual Processor Engine.* The system uses ordinary lisp objects to represent the virtual processors which are allocated as they are needed. This allows it to interact with the garbage collector to reclaim the processors as they become free.

Allocating Virtual Processors

Each PE supports virtual processors with identifiers equal to its own modulo the array size, for example on a 1K machine, PE 0 would support virtual processors 0, 1024, 2048 etc. This is different from the connection machine where virtual processors with contiguous identifiers are placed on the same PE. If a virtual processor has been allocated then it is considered to be a TACOΣ class (possibly the null class) instance and is represented by an object matching the class definition. A free virtual processor is simply represented by its identifier.

On each PE the objects representing the supported virtual processors are held in a list in ascending identifier order. When allocating virtual processors the lists are searched for free VPs before creating *new* virtual processors. After the mark phase of garbage collection, if any TACOΣ objects in the VP lists are unmarked they are replaced with their identifier, in this way the VP Engine interacts with the garbage collector to reclaim the unused virtual processors.

The virtual processor engine accepts requests for virtual processor sets of any size which it attempts to distribute across the array as evenly as possible. If the size of the requested set, n is less than the physical array size then it allocates one virtual processor from the first n PEs it finds them, as a result the set may well not be contiguous. Below in Figure 5-6 we give the algorithm for identifying

[2] There is also limited support for vectors.

want virtual processors, each on a separate physical processor:

```
for all k in parallel do
    searching[k] := true
    found := 0
    while (found < want) ∧ (searching[k])
      vpid[k] := next_vp_plane()
      if (free?(vpid[k]))
        numbered[k] := enumerate
        if (numbered[k] < (want − found))
          found := found + count
          searching[k] := false
        fi
      fi
    od
od
```

Figure 5-6: Pseudo-code for Identifying *want* Virtual Processors.

The function next_vp_plane returns the next entry in the virtual processor list on each processor, automatically creating and initialising a new virtual processor plane when the end of the list is reached. The function **enumerate** numbers the active processors and **count** returns the actual number of active processors, these are both prefix operations. Initially all PEs are searching for a virtual processor, once a PE has contributed a virtual processor it removes itself from the searching set. The allocator can be constrained to deliver a contiguous set of processors, in which case the algorithm is slightly different. On each iteration of the search we use a segmented scan operator to enumerate the *contiguous* sets of free virtual processors in the current plane. If any PE receives a value not less than the desired number of virtual processors it is a candidate for the last PE in the set – the locations of the rest of the PEs are then easy to find.

If the size of the virtual processor set requested is greater than the physical array size the allocator is less discriminating about the loading. The load is initially calculated as the minimum overestimate where each PE has the same number of VPEs:

load = (*want* + (array_size - 1)) / array_size

We then calculate how big the overestimate is:

xs = (*load* × array_size) - *want*

The load for each process is then (*load* - 1) for the last *xs* processors and *load* for the rest. The code to search for a specific number of virtual processors on each PE is much simpler than the earlier algorithm.

On each processor the identifiers are stored in order in a vector of sufficient length. A global plural space slice is allocated in which the vectors are stored, on those PEs where no virtual processors were allocated, `nil`, rather than an empty vector is stored. Thus the virtual processor allocator creates a *context* (See Section 5.1.1, page 109) which can be used as the basis of a paralation.

Creating a Paralation

To create a paralation a virtual processor set is allocated. This gives a context for the paralation. TACOΣ objects are then created for each site, the virtual processor identifiers are stored in the objects and these are then stored in the vp-lists, indicating that the virtual processors are allocated. This can be done in lisp as BLINDPEU permits access to vp-lists. Below the function `default-init` creates a TACOΣ object, setting the class and virtual processor identifier slots, and then places the object in the vp-list, which is returned by the function `vp-list`.

```
(defun allocate-tacos-object (vpid class slots)
  (let ((new (allocate-object class (+ slots 1))))
    (slot-set new 1 vpid)
    new))

(defun default-init (vpid class slots)
  (let ((tacos-object (allocate-tacos-object class vpid slots)))
    ((setter list-ref) (vp-list) (/ vpid (array-size)) tacos-object)
    tacos-object))
```

Because the paralation has still not been fully initialised `elwise` cannot be used to call `default-init`. So the `make-paralation` code must make an explicit parallel call using `pcall`. This is done in the fragment below, the function `bang` projects a singular value into a virtual processor set.

```
(let* ((ctxt-ofst (vpalloc size))
       (taco-ofst (pcall (get-bcfun 'default-init) ctxt-ofst
                         (list ctxt-ofst
                               (bang ctxt-ofst class)
                               (bang ctxt-ofst slots)) *over-loaded*)))
  (pcall (get-bcfun 'hack-context) ctxt-ofst
         (list ctxt-ofst taco-ofst) *unloaded*))
```

We consider each paralation site to be a TACOΣ class instance. To this end, the virtual processor identifiers in the paralation context are replaced with the TACOΣ instances. As well as making the

objects easier to access, the paralation now serves as a GC root for the TACOΣ objects: if a paralation is collected then the TACOΣ objects will be collected as well and the associated virtual processors will be reclaimed. Making a *simple* parallel call enables us to access and modify the overload vectors themselves. The function `hack-context` copies the vectors of TACOΣ objects into the the context vectors.

```
(defun hack-context (vpid-vec taco-vec)
  (labels ((loop (i len)
             (if (= i len) ()
               (progn
                ((setter vector-ref) vpid-vec i (vector-ref taco-vec i))
                (loop (+ i 1) len)))))
    (loop 0 (if vpid-vec (vector-length vpid-vec) 0))))
```

We need one final special function to complete the paralation, `enumerate` generates an index field for the new paralation. The virtual processors are numbered so that paralation sites which have close index values will be on the same physical processor; in this way the system is like the Connection Machine virtual processor mechanism. Below in figure 5-7 is the algorithm used to generate the index field. This also must be invoked by a simple parallel call so that it can access the overload vectors.

for all k **in parallel do**
  $index[k]$:= scan-+(length($context[k]$))
  $slot[k]$:= length($context[k]$)
  while $(slot > 0)$ **do**
    --slot
    SlotSet(*result_vector*, *slot*, *index*)
    --index
  od
od

Figure 5-7: Pseudo-code for `enumerate`

This gives us all the technology we need to create and use a paralation. A EULISP module (`plisp`) defines TEΛOΣ objects for paralations and mappings and various functions and macros which give the functionality of Paralation Lisp. This is all straight forward lisp programming and does not merit our attention here.

5.2.6 TACOΣ Operations in BLINDPEU

The technique used in BLINDPEU to represent sets of virtual processors was chosen to simplify TACOΣ operations on these sets. BLINDPEU supplies a global context which spans the entire array; lisp can be executed in this context to create and manipulate the processor collections. For example the overload vectors can be merged together to create a union of two paralations.

In a constructed paralation the index positions of the sites become secondary to their position within the paralation's structure, as this is how we expect them to be accessed. If a paralation is being used as a list, then the list order will probably be more important than the index order and these may not be the same. However we still need to have some index field for the new paralations built using TACOΣ. At the very least we need an order for printing the elements of the fields. Although any ordering would work we attempt to produce an index field which has some bearing on the structure of the paralation. For the TACOΣ constructor operators the order of the argument paralations is used as the basis for the index order. Thus the newly allocated site has index position 0, followed by the sites of the first argument paralation, and so on.

Producing a meaningful index field for a paralation generated using `connected` is rather harder than the construction case. Currently the distance from the root node is used as the basis for the order, i.e those sites with smaller index positions are those which are nearer the root site. To identify the connected sites a wave is propagated out from the root site. The distance of a site from the root corresponds to the number of the iteration that the site was first marked as connected. This value is associated with each site and is used by the code generating the index field for the connected paralation.

`Connected` is a good example of how BLINDPEU supports the operations needed by TACOΣ. All the code for `connected` has been written in lisp. This is possible because BLINDPEU gives access to the virtual processor lists on all processors and because BLINDPEU supplies the RPUT instruction (see Section 6.3.3). First, a context is created which contains every active virtual processor, the marker propagation code is then run in this context. Initially the root object is marked, then each marked object writes a mark to each object it points to. This process is then repeated until no new objects are marked. Having identified all the TACOΣ objects connected to the root they are then collected into a new paralation context.

Such as it is, BLINDPEU has proved very useful as a development system. Its basic organisation has been motivated by the requirements of TACOΣ which has allowed the various operators to be implemented in lisp. An important feature is the use of a value on every PE to indicate which PEs belong to a paralation – this makes it straightforward to write functions that can build paralations.

An alternative way of identifying processor sets is to use a segmented representation, where the processor sets are contiguous. This is used in NESL, Paralation Lisp, Connection Machine Lisp and Plural EULISP. Although very simple, it is impossible to combine processor sets in any way because in general the result will not be a contiguous segment (segments are not closed under union). The obvious advantage of segments is they are much cheaper than the system used in BLINDPEU, requiring only two words to specify the set rather than a word on every processor in the array. However some optimisations can be made to the context mechanism used by BLINDPEU: an obvious improvement would be to use a bit plane rather than a word plane to specify the context. Another possibility is to divide the array into portions, and to use a bit-code to indicate in which areas the paralation is allocated. Operations on the sets of processors would also require operations on the bit-code associated with them.

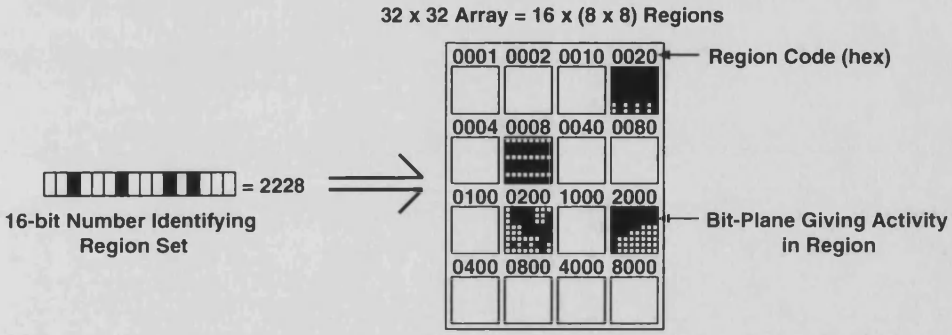


Figure 5-8: An Optimised Context Representation

Figure 5-8 illustrates this optimised context mechanism. A 32×32 array has been split into 16 8×8 regions, so any combination of regions can be represented by a 16-bit word. A plural space slice is used to indicate which individual processors within the region belong to the paralation, but the slice is only used within the regions, the same slice can be used by other paralations in disjoint sets of regions.

5.3 Nested Parallelism

Another potential problem with the representation used by BLINDPEU is *nested parallelism*. This refers to the ability to nest parallel data structures which can be manipulated using nested parallel expressions. Paralation Lisp, Connection Machine Lisp and NESL all support nested parallelism:

```
(setq fields (elwise ((n (make-paralation 5)))
                     (make-paralation (+ n 1))))
⇒ #F(#F(0) #F(0 1) #F(0 1 2) #F(0 1 2 3) #F(0 1 2 3 4))
```

```
(elwise ((field fields))
  (elwise (field) (list-ref '(a b c d e) field)))
⇒ #F(#F(a) #F(a b) #F(a b c) #F(a b c d) #F(a b c d e))
```

In the Paralation Lisp example above a `list-ref` operation is being executed on each of 15 sites. If all levels of the expression are evaluated in parallel then the 15 `list-ref` operations would be performed simultaneously in parallel.

In a similar vein Blelloch and Sabot identify two different kinds of parallelism that can be exploited when defining a parallel implementation for an algorithm. They use the quicksort algorithm to illustrate the difference between these two forms of parallelism:

quicksort(A)
  if ($\neg$ sorted(A))
    for all k **in parallel do**
      $pivot := A$[random(**length**(A))]
      $A :=$ **append**(quicksort(**collect**($A[k] < pivot$),
                quicksort(**collect**($A[k] \geq pivot$))
    od
  fi
  return A

If the array A is not already sorted then a random pivot value is chosen from A and it is split into two sub-arrays, one containing the elements of A less than the pivot value, and one containing all the other elements. The function **collect** packs the elements the boolean parameter is true for into a new array. Quicksort is then applied to these arrays and the results are appended to give an array containing the sorted elements of A.

The two possible types of parallelism in the algorithm are:

intraroutine: Operations like comparing the values to the pivot and checking the array is sorted can be implemented in parallel. This type of parallelism seems naturally suited to SIMD architectures.

interroutine: The algorithm contains two recursive calls to quicksort, each of these can be run in parallel. This seems more suited to coarse-grain MIMD architectures.

If we only take advantage of intraroutine parallelism the code will execute rapidly in the first stages, where the vectors are large, but will be inefficient in the later stages. Each invocation of

quicksort would have to be run separately and the vectors would be small. If we only take advantage of interroutine parallelism the code will perform well in later stages but poorly at first when there are large vectors to process.

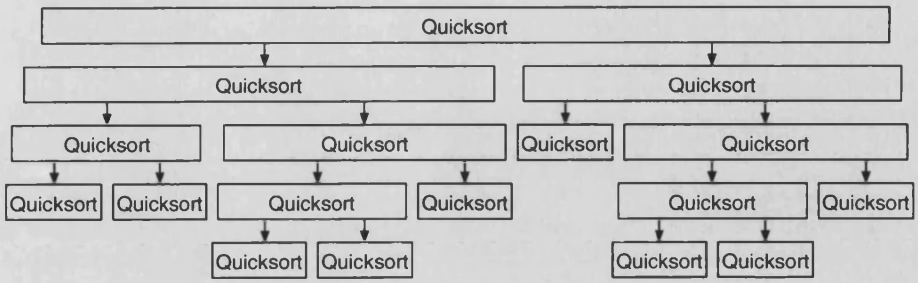


Figure 5-9: Inter and Intraroutine Parallelism in Quicksort

Figure 5-9 attempts to represent both the forms of parallelism used in quicksort. The complexity if only using intraroutine parallelism will be the complexity for the largest block, times the number of blocks, so at least $O(n \log n)$[3]. If only interroutine parallelism is used then the complexity will be that for the largest block, times the tree depth, also $O(n \log n)$. If both forms of parallelism are used then the complexity will be $O(\log^2 n)$.

```
0 (defun qsort (keys)
1   (if (sorted-p keys) keys
2     (let* ((pivot-value (field-ref keys (random (length keys))))
3            (side (elwise ((key keys)) (< key pivot-value)))
4            (sub-data (collect keys (collapse side)))
5            (sorted-sub-data (elwise (sub-data) (qsort sub-data))))
6       (expand sorted-sub-data))))
```

Above we give an implementation of quicksort in Paralation Lisp which matches the algorithm given earlier. To take advantage of both forms of parallelism our implementation of Paralation Lisp must support nested parallelism. To see this we will consider the steps in one call of `qsort` with the field `#F(7 9 2 11 19 6 12)`. In line 2, a pivot value is chosen, 9 say. In line 3 each element determines which side of the pivot it lies, by comparing itself to the pivot value (intraroutine parallelism):

```
side = #F(t () t () () t ())
```

The library functions `collect` and `collapse` collect the elements of `keys` into two new par-

[3] Guy Blelloch argues that many prefix operations can be considered to have constant complexity, in which case the complexity for each block would be $O(1)$.

alations depending on their value of `side`. The two fields are held in another new field of two elements.

```
sub-data = #F(#F(7 2 6) #F(9 11 19 12))
```

As the sub-data is held in a nested field `elwise` can be used in line 5 to apply `qsort` to both collections in parallel (interroutine parallelism). As `qsort` contains further `elwise` expressions support for nested parallelism is needed if they are to both run in parallel. The sorted fields are appended in line 6 to give the sorted result:

```
sorted-sub-data = #F(#F(2 6 7) #F(9 11 12 19))
```

```
qsort result = #F(2 6 7 9 11 12 19)
```

In the early implementation of Paralation Lisp and Connection Machine Lisp nested parallelsm was not supported and so only intraroutine parallelism was taken advantage of, i.e. each invocation of `qsort` had to be run separately. More recent versions do support full nested parallelism and NESL was specifically designed to support nested parallelism.

5.3.1 Flattening Nested Parallelism

NESL and Paralation Lisp both use compiler technology to flatten out nested parallel expressions and data structures so they can be mapped onto data parallel architectures. (The same techniques have also been used to implement a subset of the language *Proteus* [51].) The outline of the techniques given here is based on the description given for the Paralation Lisp compiler [8]. Both languages are compiled into a simple intermediate language, Paralation Lisp to Scan-Vector Lisp and NESL to VCODE. Both VCODE and SV-Lisp consist of a set of data parallel primitives which operate on vectors, e.g. `p+`, `p-and` *etc.* The important feature of the languages is their support for segmented vectors which allow a set of independent vectors to be represented and operated on as a single vector.

Vector	:	[0 1 2 3 4 5 6 7 8]
Segment Descriptor	:	[2 4 3]
Segmented Vector	:	[0 1] [2 3 4 5] [6 7 8]

Figure 5-10: Example of a Segmented Vector

Figure 5-10 shows how two vectors are used to represent a segmented vector. The first vector contains all the values, the second vector contains all the lengths of the sub-vectors. Many of the vector operators, like `p-+`, can be applied to segmented vectors by simply applying them to the value

vectors. The operators where elements of the same vectors interact are less trivial as the segmentation becomes important.

```
A                       : [5  1] [3  4  3  9] [2  6]
B                       : [1  0] [2  0  3  1] [0  1]
I                       : [0  3  1]
S (segment descriptor)  : [2  4  2]

p-+-scan(A)             : [0  5] [0  3  7  10] [0  2]
p-permute(A, B)         : [1  5] [4  9  3  3] [2  6]
p-extract(A, I)         : [5  9  9]
```

Figure 5-11: Examples of Operators on Vectors that use Segmentation Data

If we make the restriction that our data parallel objects (fields and vectors) must be homogeneous then it is clear we can represent them using segmented vectors, but it is not obvious how this scheme could be used to handle vectors of heterogeneous data. Support for segmented vectors constitutes one part of the flattening process, by taking a nested data structure and representing it as a flat structure. A field is represented by a `pfield` structure which has slots for values and segmentation, the value slot may contain another `pfield` structure, allowing arbitrarily nested vectors to be represented by nested `pfield` structures.

```
#F(#F(7 4) #F(11) #F(8 1 7))

pfield
   segdes: [3]
   values: pfield
              segdes: [2 1 3]
              values: [7 4 11 8 1 7]
```

Figure 5-12: Representing a Nested Field

When the values of a nested field/vector are stored in a single vector supporting full nested parallelism is greatly simplified. For each function the compiler generates two versions, one for serial execution and another for parallel execution. Generating the code for serial execution is a fairly straightforward translation to the destination language, i.e. VCODE or SV-lisp. But for the parallel version various extra code is added to handle nested parallel forms. If such a form is encountered the objects involved will also be suitably nested and represented by a nested `pfield` structure. The body of the code needs to be applied to the values in the next level of nesting, which are obtained by accessing the `values` slot of the current `pfield` structure. By virtue of the segmented representation

all the values in the next nested level will be held in a single vector, to which the body of the parallel form may be applied fully in parallel. This process is called *stepping down*. When exiting from a parallel expression the result is wrapped with a `pfield` structure containing the appropriate segmentation data, this is called *stepping up*.

There is one major difficulty remaining, which is how to handle conditionals, i.e. how do we allow different processors to execute different branches of a program. For each branch of the conditional the compiler inserts code to pack the active segments into smaller vectors. An `or-reduce` is inserted to determine whether any segments are active for the branch and if so the code is executed on these smaller vectors. After each branch has been executed the resulting vectors are merged to give the final result. Two special functions `recursive-pack` and `recursive-flag-merge` perform these tasks and can be applied to nested fields.

5.3.2 Nested Parallelism in BLINDPEU

The nature of TACOΣ means that in general the sites of a paralation will not form a contiguous set. This means that nested parallelism cannot be supported for TACOΣ paralations using the techniques based on segmented vectors described in the previous section. However BLINDPEU is able to make more virtual interpreters become active while the interpreter is running. That is to say, the number of virtual interpreters running is not restricted to those that were active when the interpreter was invoked. This feature can be used to support nested parallelism.

Parallel Call Operator

BLINDPEU has a `pcall` bytecode which is effectively a primitive `elwise` operation. Like the parallel call in Paralation EULISP (see Section 5.1.3) it takes a function address, an execution context and a set of arguments. To make the parallel call, a set of interpreter elements must be initialised in the same way that the system is initialised before the interpreter is invoked (see Section 5.1.2). These elements will become active on the next iteration of the instruction loop when all active elements are turned on before broadcasting the instruction set. Because only one interpreter element set can be initialised at a time it is necessary to sequentialise over the set of elements executing the `pcall` bytecode. The mpl code segment below has this effect:

```
{
  plural int PEs = 1;
  while (PEs) if (iproc == selectOne()) {
    PEs = 0;
```

```
    PCall-Body
  }
}
```

The function `selectOne` returns the identifier of one of the currently active processors. The global parallel variable `iproc` numbers the PEs from 0 to *arraysize* - 1. Thus within the body of the `if` statement only one of the active PEs remains active. It then removes itself from the set of PEs waiting to be processed by setting its value of `PEs` to zero. To make a `pcall` the interpreter element must perform essentially the same operations as when the host makes a `pcall` to the MASPAR. To this end all the required information is extracted from the PE, and passed to a segment of code which is very similar to the code which initialises and invokes the interpreter (see Section 5.1.2). It is necessary to make the entire array active using the `all` statement so that interpreter elements on PEs other than the current `pcall` PE can be activated.

As the result of the `pcall` will be a collection of objects, a plural space segment is allocated to hold them and the offset placed on the stack as the return value. Although the processor now has its result, it is safer and simpler to have it wait for the parallel operation it has spawned to complete. Part of the interpreter elements state is its activity status:

`active`	*status*	*comment*
0	Dying	used elsewhere
1	InActive	The interpreter is inactive and free to be allocated.
2	Pending	used elsewhere
3	Active	The interpreter is active and currently executing.
> 3	Suspended	The interpreter is active but waiting for child processes to complete.

On making a `pcall` the activity of the parent interpreter element is set to three plus the number of child processes it has. While the activity is greater than three there are still active children and the interpreter element remains suspended.

The initialisation of each interpreter element includes pushing a completion context and the arguments onto the stack, setting various registers like the program counter and environment pointer and setting the interpreter element's completion data. The completion data is used by the interpreter element once execution has finished, it includes the identifier of the parent interpreter and also

specifies a location for the interpreter elements result. This information is used by the `return` bytecode.

Parallel Return

Since we have a parallel call we also need a parallel return. This is an extension of the ordinary return operation. The program counter in the return context is set to an unreachable location when the process is invoked by `pcall` and this value is checked for by the `return` instruction. When encountered the interpreter needs to deactivate itself and write its result back into the plural space slice specified by the completion data. These are both simple parallel operations. It is then necessary for the interpreter element to notify its parent it has completed, the code segment below performs this task. The operation is quite complicated because we can have several processors returning to any suspended interpreter element. Again we sequentialise over the active set, but having chosen a PE we then activate all the PEs which are returning to the same Interpreter Element. The processors are counted using a reduction and the sum is subtracted from the parent's activity value. Once all the parent's children have completed its activity will return to 3 (= Active) and it will automatically continue processing.

```
while (Parents >= 0) {

  parent_pe = proc[selectOne()].Parents;
  if (Parents == parent_pe) {

    proc[parent_pe].reg.active -= reduceAdd32((plural int) 1);
    Parents = -1;
  }
}
```

The code is similar to the previous segment, here `proc` is used to extract the value of the parallel variable `Parents` on a single processor and to update `reg.active` on another PE.

5.3.3 Comments

It is difficult to compare the method of handling nested parallelism used by NESL with that used in BLINDPEU, one being based on compiler technology while the other is a runtime technique. However both systems have their advantages and limitations.

The first and probably most obvious difference is that the system used in BLINDPEU is not as efficient as the flattening technique. There is an overhead which is proportional to the number of processes doing an `elwise`. That is, in the code fragment below the overhead is proportional to the size of the paralation that the field `outer` belongs to.

```
(elwise ((inner outer))
  (elwise (inner) (fibonacci 10)))
```

Despite this the technique does successfully make full parallel execution of nested parallel expressions possible. If we think back to the tree diagram of quicksort's execution (Figure 5-9), each block is executed in parallel and each layer of blocks is executed in parallel. So inter and intraroutine parallelism are being supported. The `pcall` mechanism can be thought of as associating a cost with each arrow in the diagram, and forcing the arrows themselves to be executed separately. So the runtime technique will perform poorly if `outer` is very large in comparison to the size of the fields it contains.

On the other hand as a runtime method it is very versatile and can support nested parallelism in many situations. For example it can cope with expressions like:

```
(elwise ((inner outer))
  (if (fieldp inner) (elwise (inner) (fibonacci 10))
    (fibonacci 10)))
```

where not all the elements of the field `outer` are fields. This is something that the compilation technique cannot do as it requires the fields to be homogeneous. For NESL this is not a problem because this forms a part of the language design: its strong typing is much like that in ML and homogeneous vectors fit in naturally with this. However the Paralation Model is proposed as a set of extensions for any base language and a lisp programmer may be unhappy with these restrictions.

The real problem perhaps is that extending lisp-like languages for parallel execution is fundamentally difficult due to the large number of features and the complete freedom they permit the programmer. An example of this is the question of side-effects – this can allow interaction between nested function calls which makes nested data-parallelism difficult to implement [9]. For example, in paralation lisp a singular binding can be captured within a parallel environment (see Section 4.2.1, page 90) which can be updated in parallel. It proved difficult to allow interpreter elements in BLINDPEU to update a binding in the host EULISP process. Not having the global address space of the Connection Machine, it would require sequentialised requests to the host and mechanisms to avoid unnecessary writes. This was further complicated by EULISP and BLINDPEU being different systems,

making it difficult for the two environments to interact with each other properly. Currently BLINDPEU simply copies the value of the binding to the processors in the same way that NESL does, and so does not support updates.

The design of NESL recognises and avoids these problems, being strongly typed and side-effect-free. The subset of Paralation Lisp supported by the compiler described by Sabot and Blelloch [8] is really a variation on NESL, it is effectively strong-typed and the mapping support consists of a set of functions similar to those supplied in NESL. Another notable difference between NESL and Paralation Lisp is that NESL has no real concept of a site. A paralation represents a collection of processing sites which are allocated to ensure some kind of locality. Whereas in NESL parallel operations are simply applied to vectors, and they may need to be moved to make this possible. This is also reflected in the Paralation Lisp compiler where two fields of equal length are considered to be in the same paralation. BLINDPEU represents a serious attempt to implement paralation lisp, and addresses some of the problems which NESL avoids.

As a final point the quicksort example requires paralations to be decomposed into smaller paralations and then glued back together. The implementations of `expand` and `collect` in the Paralation Lisp compiler are very efficient, more so than the equivalent implementations using TACOΣ (see Section 4.5). However the reason for this is the number of sites being operated on remains constant and the operations are simply permutations, or modifications to the nested structure of the sites.

5.4 Summary

In this chapter we have looked at some key issues in the implementation of Paralation Lisp and TACOΣ for data parallel architectures such as the *MASPAR*. The BLINDPEU system has served as a useful basis for discussing these issues and describing some of the techniques used to realise them.

The need for virtual processors is probably the most important aspect of supporting active objects. The constructive features of TACOΣ make it possible to perform set-like operations on collections of processing sites, so the method used to identify these collections needs to be suitable for these kinds of operations. It must also be able to support multiple virtual sites on each physical processing element. The method used in BLINDPEU where a context, i.e. a slice of PE memory across the entire array, indicates which PEs hold sites of a paralation, is well suited to set-like operations. BLINDPEU also illustrates some basic mechanisms for managing virtual processors, e.g. their allocation and collection.

Although TACOΣ makes it impossible to use a segmented representation like that in NESL, it is still possible to give reasonable support for nested parallelism. However there is an overhead associated

with the mechanism and further work to improve both representations and compiler technology for supporting nested parallelism would be useful. We have also seen earlier that TACOΣ is able to support many of the useful features of NESL's representation, e.g. segmented scans (see Section 4.5).

We have now demonstrated that the allocation, construction and computation features of TACOΣ can all be implemented realistically. There are some drawbacks, but these are balanced by TACOΣ making alternative techniques practical. In the next chapter we will see this is also true for the communication features of TACOΣ.

Chapter 6

Implementations for Communication

In the previous chapter we looked mostly at the computation aspects of implementing data-parallel languages, in particular how one should support virtual processors. In this chapter we will look at implementations of the communication mechanisms in these languages, and once again we will find that the support of virtual processes proves to be the key issue. However this is a problem intrinsic to inter-virtual-processor communication and not to a particular communication paradigm. We will first consider how inter-processor references are constructed and then look at actually moving objects between processors.

6.1 Constructing a Connection

By "constructing a connection", we refer to the mechanism by which a processor identifies the processor with which it needs to communicate. At the language level we can group the mechanisms into two classes:

Primitive mechanisms where processors are simply specified by their index position with respect to some set, e.g. `permute` in NESL.

Abstract mechanisms where processors are identified by relations between objects allocated on them – mappings and β fall into this category.

Quite clearly the primitive mechanisms will prove much easier to implement than the more abstract ones. In NESL where the vectors are contiguous segments of processors it is trivial to identify a processor from its index position. It is much harder to identify efficiently a processor from some arbitrary object it contains. This is further complicated by the mechanisms permitting collisions which are resolved by combining the values as they occur. The mechanism used must also take

account of this — especially as collision order can be important. So the implementations also fall into two categories:

Simple: Where a processor can be identified simply by knowing its position within a contiguous collection of processors and where the collection begins.

Complex: Where some associative look-up mechanism is required that allows a processor to be identified by its contents. Naively this will be a search, but parallel architectures often lend themselves to efficient parallel implementations.

TACOΣ lies partially between these two classes — the processors are specified by giving their index position within a paralation, but as the sites making up a paralation may not be contiguous, `make-target` cannot simply use arithmetic to convert an index to a processor identifier. Before discussing the implementation of `make-target` we will look at the strategies used for implementing `match` and β.

6.1.1 Mappings

In his discussion of the implementation of `match`, Sabot spends much of the time describing serial implementations based on tables. A table is used as a collection of rendezvous sites. Each value in the source field is written into the table under its key, if a value has already been written to the table then it is combined with the new value. Each element of the destination field then extracts the element stored in the table under its key. He goes on to consider various modifications that can be made to this basic algorithm to make it suitable for parallel execution. One such improvement is the use of canonical mappings, which he describes as follows:

> Suppose the key fields are K1 and K2 (it does not matter which is the the source or destination). Each key in K1 is labelled with the index of its first occurrence in K2, or `nil` if it is not needed because it was not found in K2 (a key is only needed if it occurs at least once in *both* K1 and K2). Next each key K in K2 is labelled with its label in K1, or `nil` if it is not found there. The labels of K1 and K2 represent the canonicalised mapping.

The process of canonicalisation converts the fields given to `match` into an equivalent pair of fields which, in general, will be easier to work with. Thus in a possible implementation `match` may create a `mapping` structure containing the canonicalised fields, further calculations would then be performed by `move` when the mapping was used.

```
(match '#F(a b c d b) '#F(b a a z a))
⇒ <mapping :to-key #F(0 1 () () 1)
           :from-key #F(1 0 0 () 0)>
```

To create a canonicalised mapping he uses much the same process as his serial implementation of `move`, this time the table is used to associate labels with keys. The advantage of the canonicalised maps is they transfer some of the work in `move`, which may be used several times for one mapping, into `match`, which is only called once. The advantage of the labels is they identify a set of contiguous locations which can be used as rendezvous sites, these could be vector elements or individual processors. Each source element writes its value to the rendezvous site specified by its canonical label, with the collisions being combined. Each destination element then reads a value from the site its canonical key specifies. So canonical maps allow the processor array to be used as a lookup table. It is appropriate to mention here the *rendezvous mechanism* used in Connection Machine Lisp (see also Sections 2.3.1 and 2.3.3).

A xapping is a collection of ordered pairs, *key* → *value*. Every object which is used as a *key* is allocated a unique processor and a *key* → *value* pair is stored as the *value* and the identifier of the unique processor allocated for the *key*. This unique processor is known as a rendezvous location, and can be used by β when given two arguments in the same way as the labels in canonicalised fields. Of course to associate every *key* object with a unique processor also requires a table lookup mechanism. However the Connection Machine Lisp strategy cleverly hides the use of the table in the allocation phase, which will often be an inherently slow process anyway because of the speed of communication between host and processor array.

It is clear that the *table look up* is the key issue in `match` and `make-target` and we will now look at this particular aspect of the process in more detail. Sabot describes a parallel lookup mechanism as follows:

> Therefore, table look up can be implemented by appending the sequence of keys being looked up to the sequence of keys that make up the table, and then sorting the resulting collection. After the sort, each table key will immediately precede a contiguous group of identical keys that are trying to perform a lookup. A segmented prefix of `arg1`[1] propagates the necessary table data from each table key to the lookup keys. Finally a communication operation is used to send the looked up keys and table data to their original locations.

[1] The binary function `arg1` returns the first of its two arguments.

Although rather short on detail this gives the basic form of the algorithm. It hinges on a powerful and efficient sort operation, which will need to be able to handle collections larger than the physical array size. Also, for the table lookup, this sort operation must be able to order a set of values using the labels as a sort key. Here the values being sorted will be either the table value for the given key or the identifier of a processor doing a table look up for that key. Once sorted the table value for a key can be spread across the segment of lookup PEs for that key, then each PE will send the table value to the processors originating the lookup request. This can often be done by packing both the key and value into a single word with the key more significant. After sorting, the keys will be in the correct order and their corresponding values will also associated with them.

6.1.2 Targets

A variation of the parallel look up mechanism can be used for `make-target`. However, instead of appending the sequence of keys to the sequence representing the table, each processor contributes a request, i.e. *which PE has this site*, and a reply, i.e. *this PE has this site*. The requests and replies have the same format and are packed into a 64-bit word (mpl has a 64-bit integer type called `long long`) as follows (most significant first):

field	*size*	*comment*
`elwise` id	18	To eliminate interference in nested `elwise` expressions
index	20	The index we desire a target for (paralation size $< 10^6$)
vproc	26	The virtual processor id of the source site (total VPs $< 6 \times 10^7$)

The requests and replies are then sorted, this rearranges them so that for each active paralation, the requests and replies occur in ordered contiguous segments across the array. Since each set is the same size the requests and replies are aligned. Figure 6-1 illustrates the result of sorting requests and replies when multiple paralations are active (as the result of a nested `elwise` expression).

On each processor we now have the following data:

request-index : The index field of the request

back-to : The vproc field of the request

reply-index : The index field of the reply

reply-vproc : The vproc field of the reply

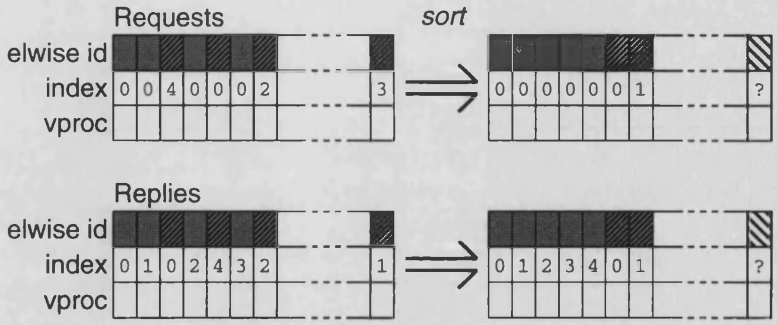


Figure 6-1: Sorting requests and replies for `make-target`

Since the values are now contiguous and ordered each processor can determine which PE holds the answer to the request it holds:

from-pe = this-pe - (request-index - reply-index)

All that remains to do is read the value of *reply-vproc* on the *from-pe* and send it to the *back-to* processor. This means the complexity of `make-target` will be the same as that for `match`. The sorting with the `elwise`-id as a primary key is an especially useful mechanism for TACOΣ as it permits data from each site of a paralation to be collected in a contiguous ordered segment of PEs, in a single logarithmic operation. Thus although the paralations are arbitrary collections of processors we can still take advantage of a segmented representation when needed.

Target Arithmetic

Another possible mechanism for creating inter-processor links is using *target arithmetic*. The idea is that if *target* is a pointer to a processor then $target + n$ will also be a pointer to a processor.

The motivation for such a mechanism is that pointer arithmetic is a standard technique used for manipulating memory. In the C programming language, memory can be allocated in blocks and the contents of the block are accessed by manipulating pointers to the block of memory. However pointer arithmetic is not meaningful in the context of data structures of small linked memory segments, since an operation on a pointer does not give another pointer in the data structure.

A similar situation exists in TACOΣ where `make-paralation` can be used to allocate blocks of processors If the underlying implementation ensures the processors are contiguous then target arithmetic can be used meaningfully to create new targets from existing targets. Where the paralation is the result of connecting individual TACOΣ instances, target arithmetic will not produce meaningful results.

In the context of lisp some may think target arithmetic is an undesirable feature. This is because being able to obtain a handle on part of an aggregate object and manipulate this handle to move to

others parts of the object is not usually permitted in lisp style languages. This is usually because it is perceived as a hole in the language through to the underlying implementation. These misgivings do seem reasonable but we should remember the paralation model and TACOΣ are intended to be applicable to a variety of languages. If we were working in the context of C or C++ target arithmetic would be a perfectly reasonable mechanism.

6.2 Communicating

Now that we are able to construct connections between processor sites we will look at how data is moved along the connections. The actual transfer of data is straightforward. What causes difficulty is resolving collisions.

6.2.1 Move

On the subject of a parallel implementation of `move` Sabot is again rather brief. As with `match` his discussion centres round a serial version, and some comments on how it could be modified for parallel execution. Again the implementation revolves round a table mechanism, though this time an array is used as canonicalised keys are now available.

A parallel implementation of `move` uses the processor array as a table in the same way that `match` did. Because the key values in the canonicalised mapping are all small integers they can be used as rendezvous locations. Each source processor participates in a *combining send* operation, where collisions are combined into a single value by a using a given binary function. Each destination processor then reads a value from the appropriate rendezvous site.

This gives the basic outline of a good parallel implementation of `move` but there are still a few details which need to be considered. An obvious question is what happens when `move` is used within an `elwise` statement? As the description stands the rendezvous sites will be shared between the separate `move`s, and this will not give the correct result. One unsatisfactory solution would be to sequentialise over the set of `move` operations to avoid this interference. A better solution is to give each `move` operation its own set of rendezvous sites. To do this we need to know how many rendezvous sites each `move` will require, this information could probably be associated with the mapping when it is canonicalised. A scan-add operation on the number of labels will specify the start of a segment of processors for each `move` operation to use for rendezvous sites.

Another important question is how do we implement a combining send? The architecture may supply some combining communication operations but is unlikely to be able to use an arbitrary (lisp) function as the combining function. It is possible to write a send operation which will detect

collisions and sequentialise over them. What happens is that only one of the colliding processors sends a value on each iteration and the combined value is accumulated on the rendezvous processor. A collision can be detected by writing a unique identifier to a processor and reading the value back to see if it arrived. Some architectures supply mechanisms for detecting collisions like the `connected` function in mpl. The complexity of this solution is $O(max(collisions))$ which will often be quite satisfactory. Another possibility is to use sorts and prefix operations in the same way that `match` does. To do this the values are sorted using their modified labels as a primary key. This will bring them into contiguous sets, and a parallel prefix operation can then be used to combine all the values. The values must then be sent to their rendezvous sites so they can be read by the destination processors. The complexity of this method should be $O(\log n)$, so if the maximum number of collisions is much greater than the log of the total number of elements, this method should be much better. However this method does require more communication and this may push up the constant in the algorithm's complexity significantly.

Support for Mappings in BlindPEu

The aspect of mappings which proved hardest to support was allowing an arbitrary combining function to be used by `move`. A simple solution is just to `cons` the colliding values into a list, and then reduce the resulting lists with the given combinator once the communication phase is completed. The drawbacks with this are it is linear in the number of collisions and the temporary list structure could be much too large to store on a single PE (this is massive parallelism remember). Thus, it is important to combine the values as they arrive and to do this `move` was written in terms of some special bytecodes. This had the added advantage that `move` could be used within `elwise` statements. BLINDPEU's implementation of mappings is a variation on the techniques described earlier and is worth a brief mention.

A key difference is that BLINDPEU does not allocate rendezvous sites in the same way. When creating a mapping the first processor that each distinct value occurs on is chosen as the rendezvous site. This avoids interference when multiple `move`s are done in parallel.

In the current implementation a site is chosen for each key in turn, but this should be done using parallel look up. The processor ids can be sorted using the keys as a primary index. This will cause the keys to occur in contiguous segments which the first processor id can be spread across using a prefix operation, each PE can then send the rendezvous site's identifier to the PE that originated its key/id pair. Figure 6-2 illustrates this process:

To make the explanation easier to follow we have not described how the rendezvous site is

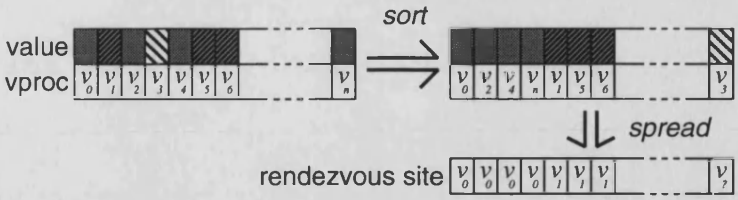


Figure 6-2: Identifying rendezvous sites in `match`

communicated to the destination processor. This is done using the method that was described for `match` earlier (Section 6.1.1). The destination processors are also included in the sort and an additional key field is used to ensure the source processors will precede the destination processors, so that the identifier of a source processor will be spread across the segment of processors.

As before we wish to pack the necessary information into as small a word size as possible, so far we have 32-bits for the value and 32-bits for the processor identifier. If we are to use this algorithm to perform multiple `matches` in parallel we must also pack a `match` id into the word to avoid interference between them. The method used for generating unique keys used by `make-target` requires too many bits for us to be able to fit all the information into a single 64-bit word. However by limiting the maximum number of simultaneous `matchs` and counting them using a prefix operation, *small, temporary* `match` ids can be generated. A possible packing could be:

field	*size*	*comment*
`match` id	7	maximum parallel `match`'s < 128
destp	1	Ensure source PEs precede the destination PEs
value	32	The keys being matched
vproc	24	The virtual processor id of the source site (total VPs $< 1 \times 10^7$)

The result of `match` is two fields, one each for the source and destination, both containing targets. The source field specifies the processors to write to and the destination field specifies the processors to read from. The combining send is implemented by a special bytecode which repeatedly attempts to send and combine the values until all the processors have been processed. On each iteration all unprocessed processors will attempt to write a value to their rendezvous processor; those processors which succeed mark themselves as finished. The written value is placed on the destination processor's stack. After the write phase all the PEs with two arguments on their stack combine them by calling the given function. The result of the send phase is a field in the source paralation where the rendezvous sites contain the combined values.

This method can be improved on by having `match` generate information that will allow a binary reduction to be performed. To do this the rendezvous site is still spread to the destination PEs in the same way as before, but for the source PEs we simply shift the processor identifiers to give each processor a *buddy*.

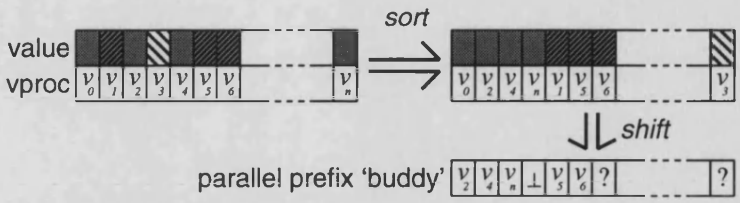


Figure 6-3: Creating parallel-prefix buddies in `match`

Now the combination phase can be performed by a `scan` like primitive (see Section 4.1.1) using the buddies generated by `match` and the values being `moved` as data. The complexity of this operation will be $O(\log(\max(collisions)))$.

6.2.2 Get, Ref and Update

We would now discuss the implementation of the TACOΣ communication primitives, but there is not a great deal to say. The primitives are very simple to implement as they are both atomic and independent. They are atomic since communication cannot really be meaningfully reduced to anything less than, "read an object from a remote processor", and "make an object available for reading". They are independent because an object can always be *read* or *made available* without the cooperation of the rest of the processor array. Strictly speaking any inter-processor communication requires the cooperation of the remote PE as it has to access its memory, however this should be all that is required of it.

The next section describes mechanisms for physically moving objects between processors. A description of BLINDPEU's approach is given and this is essentially a description of the implementation of the TACOΣ communication operators.

6.3 Moving Data

So far in this chapter we have discussed the implementation of communication primitives with the implicit assumption we can transfer data between processors. We will now take a closer look at how this is actually done, beginning with a brief discussion of possible methods and then going onto look at the approach used by BLINDPEU.

6.3.1 How Should Data be Moved?

In BLINDPEU all communication is done by copying objects between processors. An alternative to this, that would be more in keeping with spirit of lisp, would be to simply pass a reference to the object, creating a *remote pointer* of some kind. In this way the processor array would form a single, global address space.

Remote pointers are often used in distributed lisp systems [52, 50, 49] where the mechanisms and protocols needed to support them are well understood. However these systems are usually run on multi-computers or collections of workstations and our feeling was that a remote pointer mechanism would be inappropriate on a SIMD machine.

On a MIMD platform each node can execute its own instruction stream and resolve remote pointers as necessary. This will require the cooperation of the processor holding the object but will not affect the rest of the processors. However on a SIMD platform *all* the processors will have to halt when a remote pointer is dereferenced. This would probably be acceptable if the dereferencing of remote pointers was synchronised and evenly distributed around the array. But is seems inevitable that several processors will want to access the same remote value, or possibly different values on the same processor. These accesses would have to be sequentialised, as a result a serious bottle neck will occur at processors holding values used by other processors, and this will not only halt the processors involved, but all the other processors as well. The situation could be improved by synchronising these operations across the array, but in general, it will be difficult to predict when a remote access will be made. This means that the array may have to halt several times to resolve these accesses even though they could have all been done simultaneously. It also seemed that remote pointers would increase communication frequency – to access a slot in an object on another processor would require two communication operations - one to get a remote-pointer to the object and then another to access a slot in the object it pointed to. If the entire object were copied across then it could be accessed locally.

Copying values between processors also has its drawbacks. Chiefly it is expensive in time and memory. Rather than an object being allocated on one processor it may be duplicated over the entire array using up memory. To copy an object requires *building* a copy of it on the destination processor, allocating the object and the objects it contains, which is time consuming. Another problem is how do we copy very large or self-referential objects?

It is difficult to say which system is better as both have advantages and some applications will benefit from one and not the other. For example any number crunching, array based algorithm will benefit from using remote pointers if floating point numbers are immediate data. Perhaps the most

practical solution is to supply mechanisms, using whichever system is best suited to the data and application. This could be a language feature or the implementation may make the decision itself. Time did not permit us to experiment with different mechanisms as having a usable prototype was the expedient requirement. We now describe the mechanism used in BLINDPEU and how it interacts with the virtual processor engine (see Section 5.2.5).

6.3.2 Moving Data in BlindPEu

To copy an object between processors BLINDPEU encodes it into a string of bytes, copies the string to another processor where a copy of the object is then built from this description. The object is encoded by recursively walking over the object and writing a sequence of bytes for each object that makes it up. Thus the structure of the object is implicit in the sequence of objects in the description. Each object is encoded as follows:

field	*bytes*	*comment*
type	1	The type of the object, int, vector etc
size	1	This is in words, equivalent to the number of slots for aggregate objects
data	4	Either a float or an integer, the contents of aggregate objects will be other objects

The encode phase uses pointer-reversal to walk over the object as it is encoded, this means no additional space is needed other than that for the description string. When the copy of the object is being built a stack must be used since allocation may cause a garbage collection which also uses pointer reversal. The object is effectively built by a small, stack based, bytecode interpreter which interprets the description string.

6.3.3 The Ref and Update Instructions

BLINDPEU has an instruction for each of the functions `ref` and `update` called GET and PUT respectively. The instructions copy objects between virtual processors via special locations associated with each virtual processor.

Each processing element has a *global* vector called the *vp-vector* which is shared by all the virtual processors it supports. Each virtual processor has a specific location within this vector given by (*vproc-id* / *array-size*). These locations correspond to the *sites* associated with each virtual processor that are visible to other processors (see Section 3.5.3). Previously this vector had been a list, mirroring

the way the virtual processor engine keeps track of virtual processors (see Section 5.2.5). The use of a list though meant searches were required during communication which was inconvenient. Currently the vector is a valid lisp object which is allocated from the static heap (see Section 5.1.1) at startup time. This has the advantage that it is at a fixed location and mpl code can access it easily, but it cannot be reallocated so it effectively puts an upper limit on the number of virtual processors per PE (10).

The PUT instruction encodes the object and places the resulting byte vector in the appropriate location within the vp-vector as a lisp string object. It remains there until another another PUT instruction for that virtual processor overwrites it. The GET instruction copies the encode string from the appropriate slot of the remote vp-vector and then builds a copy of it. To read a value from the virtual processor *vproc* requires the following steps:

1. The physical remote processor is given by (*vproc* mod *array-size*).
2. The slot in the remote *vp-vector* is given by (*vproc / array-size*).
3. Reading the contents of this slot gives the address of the encode string on the remote processor.
4. The encode string is copied into a local *scratch* space and the build interpreter invoked.

In earlier versions of TACOΣ the values associated with each virtual processor were not persistent and only existed for one GET operation. This proved much harder to implement, chiefly because of space restrictions. Each PE currently has 64 bytes of scratch space, which seems adequate for most tasks, but for the GET instruction, several processors need to encode an object and this seemed to suggest increasing the scratch space to match the maximum number of virtual processors, this would use 6 of the 16k of memory on each PE! Copying the objects into the heap can still use up to this much memory, but it is reclaimable and for the most part encoding strings are much shorter than 64 bytes. Another possible solution is to encode each object in turn and let each active virtual interpreter try to read its value. This meant a lot of collisions had to be detected and iterated over and the resulting code was lengthy and verbose.

The Pull Instruction

The simplicity of the final version of GET also motivated an additional bytecode for supporting `move`[2].

`Move` consists of two phases: the combination phase which is done within the source PEs, and the communication phase where the combined values are read by the destination PEs. In the initial

[2] In general we want to avoid adding bytecodes in this way, but one of the purposes of the prototype is to allow us to add support easily for new language constructs.

implementation the result of the first phase was a field of the combined values. An additional step was added to encode these values and return a field of the encode strings. As the combine phase executes in *over-loaded* mode the resulting plural space slice contains vectors of strings. These vectors are similar in appearance to the *vp-vectors*, except there isn't a location for every virtual processor supported by the physical processor. `Match` was modified so that rather than giving the destination sites a virtual processor identifier to read from, the physical PE and the position in the overload vector were generated instead.

The new instruction PULL accepts a plural space offset, processor id, and an index into the overload vector. The remote value can be read in the same way as GET but an additional indirection is needed to find the address of the overload vector on the remote processor from the plural space offset.

The RPUT Instruction

BLINDPEU also includes a remote put instruction called RPUT. This is similar to PUT except it places the encode string in the vp-vector slot associated with another virtual processor. This instruction was supplied so that `connected` could be written in lisp. For one PE to place an encode string into the vp-vector slot of the virtual processor *vproc* requires the following step:

1. The physical remote processor is given by (*vproc* mod *array-size*).
2. *All* PEs set a variable *pe* to `-1`.
3. The following values are written to the remote processor:
   - Virtual processor identifier → *slot*
   - The local physical processor identifier → *pe*
   - Encode string length → *len*
4. The value of *pe* is read back to determine whether the write succeeded (fortunately the `router` construct in mpl is deterministic) and if so, the object being sent is encoded into the scratch space. These processors then mark themselves as finished.
5. *All* processors test for *pe* >= 0 and if true:
   - Allocate a string object of size *len* and place it in slot (*slot* / *array-size*) of *vp-vector*.
   - Copy the contents of scratch on the PE *pe*, into this string object.
6. Repeat until all processors finished.

The remote processor can access the object written to it by reading its own communication site using `(ref (make-target ()))`.

The RPUT instruction ignores collisions, i.e. if several values are written to the same virtual processor, then they will be written one after the other and only the last value written will be *seen* by the destination processor. It does handle collisions at the physical processor level since values going to the same processor may be for different virtual processors. This technique could also be used to detect collisions on virtual processors. Only writing one value rather than all the colliding values will make many RPUT operations much faster.

6.4 Summary

In this chapter we have looked at strategies for implementing the TACOΣ communication primitives and the mappings of Paralation Lisp. This gives us a good basis for further evaluating the utility of the TACOΣ operators. In Chapter 4 we looked at various examples where active objects were useful for defining and performing inter-processor communication: We now need to consider if they can be realistically implemented and how they compare to other mechanisms.

Targets are created by specifying the index position of the destination processor within its paralation – in this much they are similar to the communication primitives of NESL such as `permute`. However the nature of TACOΣ means the sites of a paralation will not, in general, be in order or contiguous. Thus unlike NESL, we cannot determine the processor where a paralation site resides simply by adding its index to the start of the paralation's segment of processors. However `make-target` can be implemented efficiently and in parallel using the same techniques used for `match`. This means that `make-target` has the same complexity as `match` though it does not seem as powerful. But this is balanced by the utility of being able to construct paralations and create targets matching the problem structure in a natural way. Also targets are more versatile than mappings as they can be moved between processors themselves creating new communication patterns without the use of `make-target`; the buddy algorithm (See Section 4.1.1) is a good example of this.

The TACOΣ communication primitives are very straightforward to implement. Further, this simplicity means their associated costs are the same as those of the same operations in the underlying architecture, i.e. the cost of `ref` is essentially that of a remote read. That the primitives could be implemented for the virtual processors of BLINDPEU in a straightforward way is a good example of this. One of the reasons for this is they do not require synchronisation. This makes the primitives also suitable for loosely connected processors as well as tightly connected systems like the Connection Machine.

In contrast `move` is a much more complicated function to implement and requires a high degree of synchronisation between the processors. This of course is because it is much more powerful than `ref` giving the programmer a simple way of handling collisions. A send operator with an arbitrary combining function seems a useful operation to have but we have seen (Section 4.4) that TACOΣ structures can be built to handle collisions efficiently. In addition, mappings are often much more powerful than needed, choosing 1 from n will be a $O(\log n)$ operation, whereas the RPUT instruction is $O(1)$, so it does seem useful to have access to primitive, as well as powerful high-level, communication operators.

In conclusion the TACOΣ primitives are all realistic functions to supply and also have some advantages that make them a useful alternative to mappings even outside the context of active data structures.

Chapter 7

Future and Related Work

In this thesis we have applied some of the ideas found in traditional object systems to an area of parallel programming. This was motivated by the observation that a specific class of parallel architectures, namely the massively parallel architectures like the Connection Machine and *MasPar*, could be viewed as coarse grain, active memory. A review of the languages currently available for these architectures showed that although they gave good control of the machines, they did not represent active memory programming languages. To redress this situation we have designed the active object system, TACOΣ, which uses familiar programming technology to handle aspects of parallel programming. TACOΣ hides the details of processor allocation and the construction of inter-processor links in the same way a conventional object system hides the details of memory allocation and the creation of pointers.

Although a great deal has been learnt from this work there is still much more experimentation with the language definition and implementation that can be done. The immediate contributions of this work are discussed in the next chapter. Here we discuss further work that can be done with the model and compare it to some existing systems that provide concurrency through objects. The chapter is divided into three sections: extending and improving the language model, directions for improving the implementation and a comparison with object-oriented concurrent languages.

7.1 Extending the Model

The definition of TACOΣ given here has arisen from a process of experimentation with the language and its implementation. It is, of course, difficult to say when this process is complete. The system is presented here in the state at which we felt it met the requirements of active memory programming. However there are still many refinements that can and need to be applied to the model. For example,

it would probably be better if both active and inactive objects were supported by a single object system. Data structures could then be constructed from both active and inactive objects reflecting dependencies between the objects. Where there is potential for concurrency the data structure could be constructed from active objects which could then be processed in parallel. To be able to merge the two systems in this way will no doubt require refining the active object system further and applying other object-oriented technology to active objects. This seems likely to be a rich area for further work and in this section we will outline some of the possibilities.

7.1.1 Lose the Targets

During the development of TACOΣ, targets and their support were worked on first with the general requirements of TACOΣ in mind. TACOΣ was then implemented using targets and lisp objects with the aid of some hooks on to the Virtual Processor Engine. The advantage of this was that most of the system could then be implemented in Lisp which simplified much of the development process.

However having built and used the system and been able to formulate a clear interpretation of the model it seems that in many ways targets are redundant. As we consider the active-object instances to be the actual paralation processing sites the objects implicitly specify a site and so represent the same information as targets. Below we repeat some of our earlier examples to illustrate how the objects can be meaningfully used instead of targets. First consider building the structure and accessing structure slots:

```
(setq p (pcons 'but-two (pcons 'but-one (pcons 'last ()))))  ;(from page 65)
⇒ #F(0 1 2)
(elwise (p) (pcdr (structure)))  ;(from page 67)
⇒ #F(#<plist> #<plist> ())
```

The contents of each `pcdr` slot is a *pointer* to the next site in the collection. Rather than a target, the object associated with the site, a `#<plist>` instance, indicates the site in question. As the objects represent the same information as targets we can redefine `get` so that objects are used instead of targets.

```
(get (elwise (p) (pcdr (structure))) p)
⇒ #F(1 2 ())
```

This generally seems a much cleaner and self-consistent model of active data structures, the problem is how could it be implemented. Because we view the objects as being the processing sites it seems unwise to allow them to be duplicated between processors; for one reason it will complicate the

garbage collection process for these objects and make it harder reclaim their associated processors. In consequence when `pcdr` returns the remote `plist` object it cannot be a copy, making it necessary to use inter-processor references. We can augment the address space to include an address which indicates a processor, and hence the object representing that processor. We have discussed before the dangers of a global address space and inter-processor references (section 6.3.1). Here though, it would be restricted to a single special class and any contention would simply be that which would have occurred anyway using the existing communication forms.

Implementing active-objects and references to them in this way has an added advantage: all the slots of an active-object become effectively *visible* to other processors. This would allow us to discard the rather clumsy `ref` and `update` functions and their implicit processor slot, instead we can simply use the active-object slots and accessors. So in our current example:

```
(elwise (p)
  (let ((next-pcons (pcdr (structure))))
    (if next-pcons (pcar next-pcons) 'no-data)))
```

Each processor applies `pcar` to the next `pcons`-cell in the list, which will be data on a remote processor so what happens? It seems sensible to adopt the strategy that if the result of referencing an active object is a remote inactive object, then the object will be copied to the processor accessing the slot.

```
⇒ #F(but-one last no-data)
```

This neatly gives us our copying, communication operation while the individual processors retain a degree of independence.

We also need some method of creating a reference to an active-processor from its index position, i.e. a replacement for the function `make-target`. We can simply replace `make-target` with some kind of `get-active-object`. However a more interesting possibility is to augment `structure` so that it can identify an object based on some expression, for example:

```
(structure (= (here) 2))
```

would return the object at index position 2, expressions based on the slots and class of the objects could also be used.

Thus, implementing a limited global address space would add to the general consistency of the object system and remove some of the less desirable mechanisms. This would give an overall cleaner interface to the active objects though it wouldn't add to the intrinsic functionality of the model.

7.1.2 Generic Functions

Generic functions are a common feature of object systems which we have not explored in the context of TACOΣ. In EULISP, having defined a generic function, methods can then be added which define the functions behaviour for specific classes of argument. So we could define the behaviour of a function when applied to a field:

```
(defmethod negate ((o field))
  (elwise (o) (negate o)))


(negate '#F(3 -5 -7 9))
⇒ #F(-3 5 7 -9)
```

Here we simply apply `negate` to each element of the field, where once again, depending on the argument's class, the correct method will be selected and applied. This gives a clean way of mapping a function over a nested field. More interestingly we may define methods which behave according to the class of the active-objects as well as, or instead of, the field contents:

```
(defmethod do-graph-node ((o field))
  (elwise (o) (do-graph-node (structure o))))
```

Thus `do-graph-node` can be recursively mapped over structures of active and ordinary classes. This could be useful if we had a structure where some sections are independent (and can be evaluated in parallel) while others must be executed in a specific order (i.e. serially). Defining a single function which behaves differently depending on where it is executing is reminiscent of `fork` in C, thus the model encompasses another style of parallel execution.

Another interesting aspect of generic functions is the `call-next-method` form. This gives a simple way of defining behaviour for a class that is the same as that of its super-class with some additional local code. This can be a very useful property if this style of code is being executed on a SIMD machine. The class hierarchy gives an order to execute each method so that the set of participating processors will be maximised. Naively, such code could be compiled as switches on type and code fragments containing calls to the super-methods, as a result the complete hierarchy of `call-next-methods` would have to be re-broadcast for each different sub-class in the initial switch. The hierarchy of methods makes the the task of Common Subexpression Induction [21] trivial, since the root method is common to all objects, the sub-method is common to all its sub-classes, and so on. Thus for some code, i.e. not overly complicated, it may only be necessary to broadcast each method once. This is of interest since it is not a programming style we would immediately expect to

be suitable for SIMD architectures.

7.1.3 Access to the Structure

In section 7.1.1 we saw that the model could be made cleaner and more self-consistent by giving users access to the structures and using them to define the inter-processor links directly rather than using the current system of targets. Earlier, in Section 3.5.3 (page 74), we voiced concerns about giving the user access to these objects as it gave too much control. Most of the apparent dangers are in fact resolved by a better implementation.

Another interesting aspect of giving access to the function `structure` is that we may also apply an updator function that allows us to change the current active object. Clearly some restrictions need to be placed on such an operator to ensure that an active object is always given as the argument.

By replacing the structure in this way we can change the structure of the processors in a single operation. Data structures are often reorganised to suit a different problem, having defined different structures for the processors (`setter structure`) would allow us to flip between them as needed. The problem is that this is a highly non-functional operation and makes it possible for bits of a structure to be changed giving the programmers ample opportunities to hang themselves if they put their minds to it.

This operation becomes much more interesting when we consider the active objects to be the processing sites, rather than objects associated with processors. Changing these objects now has a rather different meaning. If a process executing on some active-object changes the active-object, it will then be *executing* on a *different* processor. This gives the model an interesting handle on process migration.

The details of how such an operation could be meaningfully supported will require further work. But one obvious question is "what happens to the result?" This aspect of changing the structure makes it seem less attractive, starting off processes wherever we please and not worrying about the results seems an inherently bad situation. An alternative to updating `structure` could be to treat it in the same way as a lexical binding, this then could be masked by creating local definitions of structure. A possible control form for this could be:

(`let/structure` *active-object body*) → *obj*

Within this expression the object returned by `structure` will be *active-object* and hence *body* will be executing on a different processor. On exiting the expression the previous value of `structure` will become visible and so the expression will continue to execute on that processor.

We have explored the possibility of migrating processes in this way simply because `structure`

had the appearance of a slot (or binding) that could be updated (or masked) – we were not setting out to support process migration. As a result, the uses for the construct are not immediately obvious. However we can envisage a situation where rather than pulling multiple values from another active-object and using them in some computation, it would be simpler to perform the computation on the active-object and return the result!

This now means we are unable to change the structure of the processors in the way described earlier. This could be made possible by having a `change-class` style operation like that in CLOS [10, page 313]. This would allow us to change the properties of an object without changing the actual object and hence the processor. This seems an unsatisfactory operation and another method would be desirable. This is perhaps a further indication that changing the structure of the processors in this way is an undesirable feature. Since different patterns of connectivity can be implemented with extended object definitions the process migration interpretation of `(setter structure)` seems the more useful and interesting extension.

7.1.4 Meta-Object Protocols and Reflection

A Meta Object Protocol (MOP) [36] allows the representation of objects to be redefined. In general there seems to be no reason why any method for making an object system more powerful and expressive cannot also be used for an active-object system. We may also be able to use the MOP to integrate the TACOΣ operations seamlessly with the existing object systems. In essence, a MOP allows us to define how objects are allocated, initialised and accessed, thus we should be able to define a *proper* new meta-class, that uses active-primitives for these operations. A MOP which gave control over generic function dispatch would also be of use for the extensions described in Section 7.1.2. Below we give two code fragments taken from *The Art of the Metaobject Protocol*, these illustrate that the necessary hooks should be available to define an active meta-class.

```
(defmethod allocate-instance ((class standard-class))
  (allocate-std-class                                    ;allocate active class?
   class
   (allocate-slot-storage (count-if #'instance-slot-p ;allocate active slot storage?
                            (class-slots class))
                          secret-unbound-value)))

(defmethod slot-value-using-class ((class standard-class)
                                   instance slot-name)
  (let* ((location (slot-location class slot-name))
```

```
      (local-slots (std-instance-local-slots instance)) ;remote slots?
      (val (slot-contents local-slots location)))       ;copy needed?
  (if (eq secret-unbound-value val)
      (error "The slot ~S is unbound in the object ~a." slot-name instance)
      val)))
```

We have used these CLOS-style examples on the basis of it being a well known example. However given the need for compilation and efficient execution of code manipulating active objects a MOP such as that in ΤΕΛΟΣ would probably be more suitable. The ΤΕΛΟΣ MOP tries to balance efficiency and extensibility [13] by observing (among others) the following rules:

- Distinguish between development and execution user requirements
- Distinguish between compile-time and run-time dependencies between modules.
- Pay efficiency costs at load-time rather than run-time.

All these features make the possibility of using a Metaobject protocol to control active objects more feasible, as they will help to reduce interaction between the host, where the object system kernel is based, and the processor array where the objects are allocated.

What would be more interesting is a MOP that gave us control over the *active nature* of the objects. How this could be done is not immediately obvious, but it might be possible by having an `evaluate-method` for the active objects. So instead of objects simply representing sites where code is executed, code would be executed, perhaps implicitly, by applying a function to a collection of objects. Additional methods could then be added to alter how objects deal with such *evaluation requests*. For example we may be able to define objects which behave as multiple objects, thus modelling overloading of processors.

This could prove to be an important enhancement to the active object model allowing it to encompass other styles of parallelism. In a similar vein to Meta-Objects are reflective systems, but it is difficult to see how reflection [59] can be interpreted in terms of parallelism. None the less, the language extensions that reflection can make possible would certainly still be of use, and no doubt of interest, in the context of TACOΣ.

7.1.5 And What of `Elwise`?

The extensions and modifications we have discussed so far have all enhanced the active memory model of TACOΣ. The active data structures now resemble their serial counter parts closely, support

similar operations and possibly can invoke execution on each other. This leads us to wonder if `elwise` is still the best method of executing code on collections of active objects?

Certainly there is still the need to execute code on a collection of sites, as explained earlier in Section 3.5.2, page 70. Perhaps the problem is not so much specifying code to be executed on a collection of sites, but the fact the collection remains a fixed size by virtue of it being a paralation. It may perhaps be better to make the task performed by `connected` implicit in any `elwise` expression.

In Sections 7.1.3 we discussed a possible way of transferring a process to another object, but this meant results could be found on processors not actually in the paralation. This could be resolved by collecting all the results into a new paralation once the computation had completed. If we also added some mechanism for processes not to return, then the site would not be present in the collection of results. Such a system would be well suited to programs using multi-set transformations, a novel and inherently non-serial style of programming used in languages like GAMMA [3, 4] but which is not well suited to Paralation Lisp, since paralations are immutable objects.

By doing this we would reduce the status of *the paralation* in the system considerably, instead any collection would identify a processor set. These collections could then be used to invoke execution on those processors. This greatly enhances the flexibility of paralations without compromising their locality properties. It will probably mean that many different paralations will exist at any time and this could be expensive in terms of memory (see Section 5.2.6, page 5.2.6). This could be solved by using an extra level of indirection and contiguous segments could be used to represent fields in the same way as NESL (see Section 5.3), but each element of the segment would specify a processor and address.

The implicit collecting of results could be a major overhead, but it would probably be possible to determine from the code to be executed if the start and end sets would be the same, in which case the collect code would be unnecessary. It also seems likely that it would interact poorly with nested parallelism. One immediate problem is it would be difficult to tell when an operation had finished, since one is no longer sure who is part of the operation. This would probably make it necessary to wait for all activity to finish before collecting the results, which would have to be done for each nested collection in turn. However the richness of active data structures may reduce the need for nested parallelism. And although it may not be possible to support both systems efficiently simultaneously, it should be possible to implement a system that can support both systems well when used independently.

Thus, `elwise` will become a *start parallel execution* operation with an implicit *collect* the results on completion. This is a superset of the control `elwise` currently embodies where code is executed

on each site to completion.

7.2 Extending the Implementation

In chapters 5 and 6 we looked at various issues in supporting TACOΣ to satisfy ourselves that it can be realistically implemented. But there are still numerous areas where more work can be done on the effective support of active object systems. Some examples include:

- Supporting TACOΣ on different architectures, such as Multi-computers and distributed systems. This is of particular interest since fine grain massively parallel machines are being superseded by computers with a large number of powerful processors. The Thinking Machines CM-5 [65] has up to 1024 processing elements, each of which is a Sparc processor with up to 32 Mbytes of local memory. More recently the CRAY T3D containing up to 2048 DEC Alpha chips has become available.

- BLINDPEU behaves as a set of largely independent lisp processes with some communication operations. For the most part this is quite adequate but it is lacking in one important area: the prefix operations which require a high-degree of cooperation. For simplicity these were implemented in lisp and so their performance is rather poor, but as these are very powerful operations and form a key part of parallel applications this isn't really satisfactory. Ideally these operations would be supported by the bytecode interpreter, in the same way that prefix operators form part of the kernel of VCODE [16], the intermediate language used by NESL.

- It should be possible to define an intermediate language, or bytecode instruction set, that is able to take advantage of segmented representations of nested fields while still handling heterogeneous fields in the style of BLINDPEU.

- Size and access inference technology [17] used in NESL could also be made use of since we have abandoned the bulk synchronisation [69] of the Paralation Model. This technique relaxes the synchronous nature of data-parallel computations without modifying their semantics.

7.3 Active Objects Can't Act

Throughout this thesis we have referred to TACOΣ objects as being *active* objects. This seemed an appropriate term for objects allocated from active memory. This may have been an unfortunate term however, as there are other languages based on active objects, the so called object oriented concurrent

programming (OOCP) languages. In the last decade a host of such systems have been developed and extended, some examples include Actors, Concurrent SmallTalk, ABCL and its more recent derivatives and Orient84K. An overview of these languages may be found in [70]. These languages (in general) use objects in parallel environments, but they are rather different from TACOΣ. However there are also similarities and so a brief review and comparison is appropriate. Here we will briefly look at two specific object-oriented concurrent languages: Actors [29, 1] a key example as many of its features are common to other systems, and ABCL and its derivatives, which build on the Actor model to give better functionality. Having given an outline of the OOCP languages we will then discuss how they compare to TACOΣ.

7.3.1 Actors

Actors are independent, self-contained computational agents, each having a conceptual location, its *mail address*, and a *behaviour*. Actors interact with each other by sending messages. This communication is asynchronous, message delivery is guaranteed and will occur within some finite, bounded delay. An actor can send messages to any other actor of which it knows the mail address – these actors are known as a *acquaintances*. Messages can include mail addresses of actors, so the interconnection topology of an actor system is dynamic. The behaviour of an actor defines how it responds to different types of messages, this response may cause one or more of the following actions:

1. Creation of a new actor.
2. Alteration of its behaviour and acquaintances.
3. Transmission of a message to an existing actor.

Below we give the behaviour definition for an actor which behaves as the node of a stack. This example, taken from [1, page 41], is given in the minimal actor language SAL (Simple Actor Language), which has an algol-like syntax.

```
0  def stack-node(content link)
1      [ case operation of
2          pop: (customer)
3          push: (new-content)
4      end case]
5  if operation = pop ∧ content ≠ NIL then
```

```
6       become forwarder(link)
7       send content to customer
8   fi
9   if operation = push then
10      let P = new stack-node(content, link)
11      { become stack-node(new-content, P) }
12  fi end def
```

This simple example illustrates the main components in the definition of an actor system. A *stack-node* has two acquaintances, its *content* and the next member of the stack, *link*. A predefined value NIL is used to mark the bottom of the stack, thus creating an actor with NIL as its *content* will define a new stack:

```
new stack-node(NIL, SINK)
```

SINK is the mail address of some actor – presumably the result would not be simply discarded. All operations on the stack will be sent to this actor, which is known as a *receptionist* actor as it is the only member of the stack system that can receive messages from outside the system.

The behaviour definition specifies what kind of operations are supported by the actor, i.e. what kind of messages it can receive. This is given in the case section (lines 1–4) which binds the message type and parameters to identifiers for use in the definition body. When a push message is received a new actor is created with the same *link* and *content* (line 10). The actor then becomes a *stack-node* with the *new-content* and the new actor (*P*) as its *link* (line 11). When a pop message is received the *content* is sent to the customer (line 7) and the actor becomes a *forwarder* to the next actor in the stack (line 8). This means any messages received by the actor will now be sent to its *link* acquaintance.

This description has given a rough outline of the nature of actors and how actor systems are defined and used. It is clear that the independent nature of actors and the asynchronous nature of their communication makes them inherently concurrent. The model is also very simple, this makes it a good basis for discussing concurrent computation in distributed systems but as a language for developing real systems it is rather minimal. In the next section we look at a system that extends the actor model, allowing real concurrent applications to be effectively constructed.

7.3.2 ABCL Derivatives

ABCL (An object Based Concurrent Language) has been the basis of great deal of work in object-oriented concurrent systems. There are obvious similarities with Actors, concurrent objects, with

behaviours specified by *scripts* which interact by message passing. But for the purposes of practicality ABCL does not adopt the approach that all concepts within a computation must be represented by objects. Similarly the behaviour may contain conventional *applicative* and *imperative* features. This is in general simpler than defining all computation in terms of objects and makes programs easier to read and write.

We will now quickly outline how ABCL programs are written, this description is based on [71] which describes ABCL/1, a distributed version of ABCL. An object is defined using the following notation:

[object *object-name*
  (state *representation-of-local-memory*)
  (script
    (=> *message-pattern* where *constraint* ...*action*...)
    (=> *message-pattern* where *constraint* ...*action*...))]

The *message-pattern* is matched with the incoming message components, if the where *constraint* is satisfied the associated *action* is invoked. In the script below, the semaphore object accepts two kinds of messages which are distinguished by containing either the symbol :P-op or :V-op, i.e. these symbols are used as message tags.

```
[object aSemaphore
  (state [counter := 1] [process-q = [CreateQ <== [:new]]])
  (script
    (=> [:P-op]  ...action-for-P-operation...)
    (=> [:V-op]  ...action-for-V-operation...))]
```

This fragment also illustrates the message sending syntax, part of the semaphore objects state is a process queue, which is created by sending a [:new] message to the object CreateQ. All message sending expressions are of this form:

[T ⇐ M]

where T is the target object and M is the message. There are a variety of different send (⇐) operators. There are two basic kinds of message:

Ordinary type, on arrival at its target this message is placed in a queue. Checks are made to see if the message is acceptable and if so the object is activated (if currently dormant) and processes the message.

Express type, on arrival at its destination the message is placed in the objects express queue. The object will interrupt the processing of ordinary messages in order to process any messages in the express queue. Once completed it will resume processing of the ordinary message unless it was instructed to abandon the message by an express message.

There are also three different sending modes:

Past type, on sending the message the object continues with its computation. These messages are denoted for ordinary and express messages respectively as follows:

[T <= M] [T <<= M]

Now type, in this case once the message is sent the object waits for a reply. The ordinary and express versions of now messages are denoted as:

[T <== M] [T <<== M]

As there is a result associated with these communication operations they can be used in expressions, e.g. [x := [T <== M]] will bind the result to a local variable.

Future type, once the message is sent the object continues with its computation. But there will be a reply to the message and a special variable is specified to hold this result when it arrives. These are denoted as:

[T <= M $ x] [T <<= M $ x]

There are many more intricacies to the language for handling various other aspects of sending and responding to messages. For example the sender of a message is always implicitly specified, though it may be given explicitly, and can be determined by the recipient using the symbol &sender.

It can be seen that there are many ideas common to ABCL and Actors. However this is a more practical than pure system, and whereas Actors give us a good basis for reasoning about concurrency, ABCL gives the tools needed to construct real systems easily. A good example of this is the reply mechanism which is quite complex to handle with actors (though not impossible) as they can receive only one kind of message.

More recently ABCL has been extended further in order to introduce reflection into the system. The first such system is ABCL/R [71] which extends ABCL/1. Here each object has a meta-object, which models/represents the object. Object and meta-objects can send messages to their meta-objects and these transmissions correspond to reflective computations. Both ordinary and reflective messages can take place concurrently.

ABCL/R has been further extended to give the system ABCL/R2 [31]. This system introduces the real-time meta-object in order to handle so called *soft real-time* programming. This is where an action must be taken when a deadline is passed. To do this objects are monitored by their real-time meta-objects which use their reflective power to change the objects behaviour when a deadline is passed.

The most recent development is RbCl (Reflection based Concurrent language) [33]. This is similar to the other derivatives but its key feature is it has no runtime kernel (as such). This permits efficient implementations of many of the reflective mechanisms.

An overview of the hybrid project which outlines the relationships of these and other systems can be found in [33]

7.3.3 Comparison

We will now examine how the objects in concurrent programming languages such as Actors and the ABCL derivatives are different from the active objects of TACOΣ. To avoid confusion we shall use the term *object* for entities in object systems such as TACOΣ, CLOS and TELOΣ and the term *agent* for the computational entities used in Actors and ABCL.

The chief difference is that agents are responsible for computation, whereas active objects are used to define structures of processors. This is because TACOΣ extends a collection oriented language which already has a mechanism for specifying computation. Rather than entities passing messages and performing computations, operations are applied to entire collections, these operations may be either computation or communication. TACOΣ provides a mechanism for structuring such collections so that communication and construction operations are simpler and more meaningful. TACOΣ also makes it possible to write object oriented code for theses collections, but this is at the program level, not the execution level.

Another important difference is that TACOΣ is an object system, and as we saw in section 7.1.4 can be integrated with an existing object system. This gives programmers an easy way to take advantage of parallelism by making their conventional data structures (or parts) active. Where a single object system handles both ordinary and active classes generic functions can be defined for both kinds of object so that they execute in parallel when possible. As such TACOΣ embeds parallelism into an object system giving the object-oriented programmer the opportunity for parallelism in a single, familiar paradigm.

The object-oriented concurrent programming languages however, though certainly object oriented, are not object systems. At least not in the same way as CLOS and TELOΣ, rather they are

agent systems. An obvious difference between agent and object systems is that everything is an object, but not everything is an agent – the agent system forms a component of the entire system. We should observe that it is quite possible to define systems where all entities, numbers, functions, cons cells etc., are agents. Whether we would/should want to do this is a question open to/for debate, but the answer will probably be that it depends on what we are doing. When modelling independent communicating entities we should use agents, but if processing the elements of a list in parallel we only need an active list.

Making the distinction between object systems and the OOCP languages is further hampered by both systems using the same terminology. The obvious example is the term *object*, which is used for both objects and agents which we have just seen are rather different kind of entities. Another good example of this confusion is the term *meta*, which is used both in object systems and the more recent OOCP languages. In ABCL/R2 each object is monitored by its real-time meta-object, or in our terminology each agent is monitored by its meta-agent. The definition of the agent is held within the meta-agent which allows the meta-agent to change the agents behaviour. In an object system such as ΤΕΛΟΣ or CLOS a meta-class is a description (in part) of a *collection* of classes. Although it is clear why both systems (legitimately) use the same term, they are being applied to rather different concepts.

Given that active objects are not the same as agents we should consider if one system is in anyway more powerful than the other. There is one obvious aspect of ΤΑCOΣ which may appear to make it less powerful than a system of agents, its read-only communication policy. There are good reasons for this limitation:

Firstly we wish to avoid parallel writes, since we then have to decide if and how collisions are resolved. In addition to being harder to implement, they are also harder to use – whether collisions are or are not handled, parallel writes require careful use. In contrast, read is a simple primitive, that can be easily used and understood.

Secondly, when considering operations performed by structure walking processes, we feel there is only a real need to read from a pointer. Computations that affect the state of a node in the structure will generally require values from structure nodes it has pointers to. These can either be read by dereferencing the pointers, or the process may move down the pointers perform some computation and then return with the desired value. In the active data structure there is no need to *send* processes along pointers as there will already be a process at that node. Hence we only need to read a value from the node pointed at, be it part of the node's local state or a result computed by the process at the node.

So far, our experience has supported the second observation, and in a system where all the components are active the write operation is not needed. However there is one disadvantage, which is that structure walking processes can propagate data both up and down conventional memory pointers. To do this with TACOΣ requires pairs of connections. This is unfortunate since it means some conventional data structures cannot be used in active memory without modification. For example in order to propagate data to the leaves of a binary tree, each node will need a back-pointer to its parent node, this pointer is redundant when a recursive tree walk is used to propagate the data.

Another potential disadvantage is that if an object is connected to many objects then its definition will need many slots, and these will have to be processed sequentially. However even if we were able to send messages, thus eliminating the need for many slots, these messages would still have to be sequentialised. Further we saw in the Connectionist networks (Section 4.4) how multiple inputs could be efficiently handled by an alternative, but not in anyway complex, data structure.

Although the read-only policy seems a restriction we can implement an actor-like message passing system using TACOΣ. To do this we take a leaf out of the Connection Machine's book and use TACOΣ objects to represent actors and postmen (delivering mail). The postmen are connected in some useful topology and are also attached to some fixed number of actors. We can then emulate the CM-2 communication cycle (see Section 1.1.2) to deliver messages.
For each cycle:

1. Each actor looks to see if its postman has anything for it, and if so reads it.

2. Each postman looks to see if its current value has been read by anyone – if so it can read a new value, either from an actor or a postman.

3. Each actor looks to see if its value has been read, if so it can delete it and write another message.

There are of course many other details which need to be determined: We may want to use this system to deliver all messages directly, or try and interleave the cycles with execution, degrees of buffering may be needed etc. The point is we can send a message to an actor, identified by its target/active object. It will not be a difficult task, not in Lisp anyway, to turn an actor behaviour definition into a table of lambda expressions that can be used by some generic actor interpreter program. So it seems that it should be possible to implement an actor-like language using TACOΣ.

This illustrates an interesting difference between active objects and agents. The description of the message passing mechanism we have given is data-parallel, i.e. all actors will take part in steps 1 and 3 at the same time and all postmen will take part in step 2. However this need not be the case, on a MIMD architecture each process can read and test values asynchronously. As TACOΣ merely

allocates and arranges processors and communication links it is architecture independent. But agents model processes, and although the topology of a system of agents can be used on any platform, the processes will be better suited to some platforms than others.
In summary then:

- Active objects are not inherently concurrent and so they are not agents.
- Inherent concurrency means agents model processes, where as active objects model processors and connections.
- Although agents are object-oriented they are not object systems, TACOΣ on the other hand *is* an object system making parallelism available via active data structures.
- TACOΣ has a read-only communication protocol, whereas agents are able to send messages. This is not necessarily a restriction.

TACOΣ allows us to build and use structured collections of processors, the so-called active data structures, using familiar technology. As the model has been refined and extended by incorporating other object system concepts we are able to encompass other aspects of parallelism. As a result, and with the help of hazy terminology, there are similarities with the object-oriented programming languages. However, because TACOΣ comes from a different direction, and has different goals and motivations, it is also very different from these systems.

Chapter 8

Conclusion

This thesis has been concerned with the design and implementation of parallel programming languages aimed primarily at massively parallel computers such as the Connection Machine and the MasPar.

The work presented has taken the novel approach of considering the *active memory* nature of these computers. By *active memory* we imagine an architecture where every storage location has some limited processing potential associated with it. Although such machines do not as yet exist, computers like the Connection Machine, with tens of thousands of processors, can be viewed as a form of coarse-grain active memory.

To identify the requirements of an active memory programming language we examined the requirements motivating the design of the Connection Machine. These are:

Requirement I : Enough processing elements to be allocated as needed in proportion to the size of the problem.

Requirement II : The processing elements can be connected by software.

We summarise these two requirements as:

Processing elements and communication links can be allocated and manipulated with the same ease as memory.

From which we form the concept of an active memory architecture, where we can create active data structures and then operate on them in parallel. Having identified our expectations of an active memory language a review of the languages for these computers showed that although they gave good control over the hardware, they did not embody the ideas of active memory programming well. The

key aspect missing from these languages being that of *building* active data structures. The languages are collection oriented and the collections have no structure, instead collections that *represent* the desired structure must be created. Further, operations on the collections, rather than their contents, such as union and intersection, are clumsy and expensive. Extensions to these languages impart structure to the collections, but by imposing the structure not by building it, they also require a lot of additional syntax.

We have presented here the design for an active object system. In the same way that the Paralation Model can extend any base language, active objects would extend any existing object system in the base language. The description given here has been based on the TELOS-like system called TACOS. Just as a conventional object system allows complex data structures to be built while hiding the details of memory allocation and the construction of pointers, the active object system allows complex structures of processors to be created while hiding the details of processor allocation and the construction of communication links. Some of the key points of interest include:

- Active data structures can be created using a familiar, object-oriented mechanism.
- Active objects encapsulate both structure and communication. Currently communication is abstracted by accessing a special slot associated with each active object and its accessors, but this could be generalised to use any remote active slot access.
- By using `make-paralation` as a multiple, parallel version of `make-instance`, active objects are appropriate for massively parallel applications where constructing active data structures on a per-site basis would be tedious and inefficient.

By giving new interpretations to existing language mechanisms we are able to introduce parallelism without the need for additional syntax. In languages like Common Lisp and EuLisp which have powerful object systems it is possible to add these new interpretations without disturbing the framework of the existing object system, indeed they supply a mechanism for precisely this task.

With the resulting object system programmers are able to take advantage of any potential for parallelism simply by making components of their data structures *active*. Parallel execution is specified via the `elwise` form of Paralation Lisp, which is similar to the *map* forms common to many languages. Although currently there is the need for some special functions such as `connected` and `project`, relatively few new constructs are needed to add parallelism to an object-oriented language and this helps retain much of the languages programming style.

Our experimentation with the language supports these observations. In general active objects are as easy to use as their conventional counterparts; active data structures can be built in an intuitive

fashion and the code to execute on the structure often has the same basic organisation as that of the structure walking code it replaces. There are two important exceptions to this:

A read-only communication policy has been adopted in order to simplify implementation and usage. As a result data can only move in one direction along inter-processor connections whereas we typically propagate data both up and down memory pointers. This means some data structures will need modification if they are to be used as active data structures, i.e. by creating the necessary two-way links. Another option is to remove the read-only restriction, this raises the question of handling collisions, which could be simply ignored or perhaps a MOP could be supplied that allows methods for handling concurrent slot updates to be defined.

Prefix operators, a powerful component of data-parallel programs, interact well with active data structures allowing many seemingly complex operations to be handled simply and efficiently. Because prefix operations induce a binary tree on to a collection, active data structures can often be simpler than their conventional counterparts: An obvious example is that a linked list may be as effective as a binary tree. For this reason active data structures may need to be reorganised to better take advantage of prefix operators. This is an aspect of building and using active data structures that is learnt by experience rather than from an understanding of conventional data structures.

We have also examined various implementation issues for active objects. The communication primitives themselves have straightforward implementations, but the task of creating references to other processors is more difficult. The complexity of creating these references is the same as the high-level communication operators in languages like Connection Machine Lisp and Paralation Lisp. However inter-processor references have much higher potential for reuse, as they can be moved around and modified on an individual basis. The need to be able to perform set-like operations on collections of processors requires a representation that precludes the use of a flat representation of nested collections. But nested collections can still be effectively supported and the enhanced functionality of active objects can also reduce much of the need for nested parallelism.

In summary:

- Active memory programming, as motivated by the coarse-grain active memory computers, is effectively realised by the active object system described here.
- The addition of active objects to a system does not significantly increase the complexity for the programmer, as familiar notation is used to express parallelism.

- Active data structures offer a simple and effective way to take advantage of parallelism, as a programmer only needs to make a data structure active, to have the opportunity of using parallelism.

- Our experience shows that using active objects is straightforward and gives code that strongly matches the logical structure of the problem.

- Support for active object systems can be realistically implemented. The cost of the various operations is no worse than their counterparts in other languages. Some aspects cannot be supported as well, but this is balanced by improved functionality and better potential for reuse.

There is still a lot of useful work to be done on both the definition and implementation of active object systems. As well as refining the existing model, we would like to introduce other features from object systems and give these parallel interpretations. In particular applying the ideas in metaobject protocols to active objects looks to be a fruitful direction for future work. With such an *Active Metaobject Protocol* it may be possible to define other styles of parallelism using active objects. It will be very interesting to see how many aspects of parallel programming can be embodied within a single object-oriented paradigm.

Appendix A

MasPar MP-1: Technical Summary

The MasPar MP-1 is a massively parallel SIMD machine with 1024 processors scalable to 16384. The system comprises five major subsystems:

The Array Control Unit (ACU) controls the processor array by broadcasting all PE instructions. It is also capable of independent program execution.

The Processor Element Array (PE Array) executes the instruction stream broadcast by the ACU on each PE, conditional on the activity status. Each PE has 16K of local memory which can be expanded to 64K. The CPU consists of a 4-bit ALU and 192 bytes of scratch RAM.

Communication Mechanisms include:

- The 8-way X network for communication with neighbouring PEs.
- The global router, which gives random PE-to-PE communication via a hierarchical cross-bar.
- Two global busses, one for broadcasting data and instructions from the ACU and one for consolidating the status responses of all the PEs to the ACU via a logical OR-tree.

The Unix Subsystem provides UNIX services to the data-parallel system, e.g. job management.

The I/O Subsystem supports high speed communication between the host and parallel subsystem.

Bibliography

[1] Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence. MIT Press, Cambridge, MA, 1986.

[2] Backus, J. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[3] Banâtre, J. P. and Le Metayer, D. Programming by Multiset Transformation. Technical Report 522, IRISA, Campus Universitaire de Beaulieu, 35042 - Rennes Cedex, France, March 1990.

[4] Banâtre, J. P. and Le Metayer, D. Introduction to GAMMA. In *Proc. Workshop on Research Directions in High-Level Parallel Programming Languages*, pages 197–202, Mont Saint-Michel, France, June 1991. LNCS 574.

[5] Blelloch, G. E. Scans as Primitive Parallel Operations. In *Proc. International Conference on Parallel Processing*, pages 355–362. IEEE Computer Society, 1987.

[6] Blelloch, G. E. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge, M.A., 1990.

[7] Blelloch, G. E. NesL: A Nested Data-Parallel Language. Technical Report CMU-CS-92-103, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213, Jan 1992.

[8] Blelloch, G. E. and Sabot, G. W. *Compiling Collection-Oriented languages onto Massively Parallel Computers*, volume 8, pages 119–134. Journal of Parallel and Distributed Computing, 1990.

[9] Blelloch, G.E. et al. *Implementation of a Portable Nested Data-Parallel Language*, pages 102–111. Proc. of 4th ACM SIGPLAN Symposium on Priniciple and Practice of Parallel Programming PPOPP, ACM SIGPLAN Notices, Jly 1993.

[10] Bobrow, D. G. et al. Common Lisp Object System Specification: 2. Functions in the Programmer Interface. *Lisp and Symbolic Computation*, 1(3/4):299–394, Jan. 1989.

[11] Bougeé, L. *On the Semantics of Languages for Massively Parallel SIMD Architectures*, volume I, Parallel Architectures and Algorithms, pages 150–165. PARLE '91, Parallel Architectures and Languages Europe, Eindhoven, Netherlands, 1991. LNCS 505.

[12] Bräunl, T. *Massively Parallel Programming with Parallaxis.* Universität Stuttgart, Institut für Informatik, Azenbergstr. 12, D-7000 Stuttgart 1, FRG, May 1991.

[13] Bretthauer, H., Kopp, J., Davis, H.E., and Playford, K.J. Balancing the EuLisp Metaobject Protocol. *Lisp and Symbolic Computation*, 6(1/2):119–138, 1993.

[14] British Standards Institution. *Programming Language: APL*, 1989.

[15] Burdorf, C. POCONS: A Persistent Object-Based Neural Network Simulator. In *Proc. SCS Western Multiconference: Object-Oriented Simulation.* Society for Computer Simulation, 1992.

[16] Chatterjee, S. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multi-processors.* PhD thesis, School of Computer Science, Carnegie Mellon University, Oct. 1991.

[17] Blelloch, G. E. Chatterjee, S. and Fisher, A. L. Size and Access Inference for Data-Parallel Programs. In *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 130–144, June 1991.

[18] Christy, C. Virtual Processors Considered Harmful. In *Proc. Sixth Distributed Memory Computing Conference*, pages 99–103. IEEE Computer Society, April 1991.

[19] DeMichiel, L. G. Overview: The Common Lisp Object System. *Lisp and Symbolic Computation*, 1(3/4):227–245, Jan. 1989. Also includes the system specification.

[20] Bi, H. Diestelkamp, W. and Bötcher, A. *MODULA-S, A Language to exploit two dimensional Parallelism*, pages 157–169. Proc. First International ACPC Conf. on Parallel Computation, Salzburg, Austria, Sept/Oct 1991. LNCS 591.

[21] Dietz, H. Common Subexpression Induction. Technical Report MP/CS-12.92, MasPar Corporation, Sunnyvale, CA 94086, June 1992.

[22] Orponen, P. Floréen, P., Myllymäki, P. and Tirri, H. Compiling object declarations into connectionist networks. *AICOM*, 3(4):172–183, December 1990.

[23] Lynne, K. J. Goddard, N. H. and Mintz, T. Rochester Connectionist Simulator. Technical report, University of Rochester, March 1988.

[24] Goldman, K. J. Paralation Views: Abstractions for Efficient Scientific Computing on the Connection Machine. Technical Report 1542, Thinking Machines Corp., June 1989.

[25] Haddon, B. K. and Waite, Q. M. A Compacting Procedure for Variable-Length Storage Elements. *The Computer Journal*, 10:162, 1967.

[26] Harper, R. and Mitchell, K. *Introduction to Standard ML.* Edinburgh University, Edinburgh EH9 EJZ, Great Britain, 1987.

[27] Quinn, M. J. Hatcher, P. J. *Data-Parallel Programming on MIMD Computers.* MIT Press, Cambridge, MA, 1991.

[28] Quinn, M. J. Hatcher, P. J. et al. *A Production-Quality C* Compiler for Hypercube Multicomputers*, pages 73–82. Proc. of Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM Press, July 1991.

[29] Hewitt, C. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323–364, June 1977.

[30] Hillis, W. D. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[31] Honda, Y. and Tokoro, M. Soft Real-Time Programming through Reflection. In *Proc. Interational Workshop on New Models for Software Architecture'92: Reflection and Meta-Level Achitecture*, pages 12–23, Nov. 1992.

[32] Hudak, P. and Wadler, P. *Report on the programming language HASKELL.* Technical Report, Yale University, Apr. 1990.

[33] Matsuoka, S. Ichisugi, Y. and Yonezawa, A. RbCl: A Reflective Object-Oriented Concurrent Language without a Run-Time Kernel. In *Proc. Interational Workshop on New Models for Software Architecture'92: Reflection and Meta-Level Achitecture*, pages 24–35, Nov. 1992.

[34] Karp, R. M. and Ramachandran, V. *A Survey of Parallel Algorithms for Shared-Memory Machines.* North Holland, Amsterdam, 1989.

[35] Snell, J. W. Katz, W. T. and Merickel, M. B. Artificial Neural Networks. *Methods in Enzymology*, 210(42):610–636, 1992.

[36] des Rivieres, J. Kizcales, G and Bobrow, D. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

[37] Marino, G. and Succi, G. Data Structures for Parallel Execution of Functional Languages. In *Proc. PARLE '89, Parallel Languages and Architectures Europe*, volume II, Parallel Languages, Eindhoven, The Netherlands, June 1989. Springer-Verlag. LNCS 366.

[38] MasPar Corporation. *MP-1 Standard Programming Manuals, Language Reference*, 1990.

[39] MasPar Corporation. *MP-1 Standard Programming Manuals, MasPar Fortran*, 1990.

[40] Merrall, S. and Padget, J.A. Plurals: A SIMD Extension to EuLisp. *Lisp and Symbolic Computation*, 6(1/2):201–219, 1993.

[41] Merrall, S. C. and Padget, J. A. Collections and Garbage Collection. In *Proc. of International Workshop on Memory Management*, pages 473–489, Eindhoven , Netherlands, Sept 1992. Springer-Verlag. LNCS 637.

[42] Merrall, S. C. and Padget, J. A. *Plurals - A SIMD Extension to EuLisp.* Bath Mathematics and Computer Science Technical Report, 92-59, June 1992.

[43] Merrall, S. C. and Padget, J. A. TPL – An Extended Implementation of Paralation Lisp in EuLisp. Technical Report 92-58, School of Mathematics and Computer Science, Bath University, June 1992.

[44] Metcalf, M. and Reid, J. *Fortran 90 Explained.* Oxford Science Publications, OUP, 1990. ISBN 0-19-853772-7.

[45] Myczkowski, J. and Vichniac, G. Parallel Programming for Cellular Automata. Technical Report TMC-16|CA89-3, Thinking Machines Corp., 1989.

[46] Padget, J. and Nuyens, G. An Overview of EuLisp. *Lisp and Symbolic Computation*, 6(1/2):9–98, 1993. The full definition is to be published by the Commission of the European Communities, and is also available by anonymous ftp.

[47] Padget, J.A., Nuyens, G., and Bretthauer, H. *An overview of EuLisp*, volume 6(1/2), pages 9–98. Lisp and Symbolic Computation, 1993.

[48] Philippsen, M. and Walter, F. W. *Modula-2* and its Compilation*, pages 169–183. Proc. First International ACPC Conf. on Parallel Computation, Salzburg, Austria, Sept/Oct 1991. LNCS 591.

[49] Piquer, J. M. Indirect Reference Counting: A Distributed Garbage Collection Algorithm. In *PARLE '91, Parallel Architectures and Languages Europe*, volume I, Parallel Architectures and Algorithms, pages 150–165, Eindhoven, Netherlands, 1991. Springer-Verlag. LNCS 505.

[50] Piquer, J. M. *Sharing Data Structures in a Distributed Lisp.* INRIA - Ecole Polytechnique, 1992.

[51] Prins, J. F. and Palmer, D. W. Transforming high-level data-parallel programs into vector operations. In *Proc. Fourth ACM SIGPLAN Symposium on Priniciple and Practice of Parallel Programming*, pages 119–128, May 1993.

[52] Lang, B., Queinnec, C. and Piquer, J. Garbage Collecting the World. In *Proc. of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 39–51, Albuquerque, New Mexico, Jan 1992. ACM Press 0-89791-453-8/92/0001/0089.

[53] Rose, J. and Steele, G. *C*: An Extended C Language for Data Parallel Programming.* Thinking Machines Corp., 1987. Tech. Report PL87-5.

[54] Sabot, G. W. Paralation lisp reference manual. Technical Report PL87-11, Thinking Machines Corp., 1988.

[55] Sabot, G. W. *The Paralation Model: Architecture Independent SIMD Programming.* MIT Press, Cambridge, MA, 1988.

[56] Schorr, H. and Waite, W. M. An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures. *Communications of the ACM*, 10:501–506, Aug 1967.

[57] Schreiber, R. *An Assessment of the Connection Machine.* Research Institute for Advanced Computer Science, NASA Ames Research Center, Mountain View, CA 94035, spring 1990.

[58] Dubinsky, E. Schwartz, J. T., Dewar, K. and Schonberg, E. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York, 1986.

[59] Smith, B. C. Reflection and Semantics in Lisp. In *Proc. 11th ACM Symposium on Principle of Programming Languages*, pages 23–35, 1984.

[60] Steele, G. L., Jr. *Common LISP: The Language.* Digital Press, 1984.

[61] Steele, G. L., Jr., and Hillis, W. D. Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 279–297, 1986.

[62] Steele, G. L., Jr., and Hillis, W. D. *Data Parallel Algorithms*, pages 1170–1183. Communications of the ACM, Dec 1986.

[63] Steele, G. L., Jr., and Wholey, S. *Connection Machine Lisp: A Dialect of Common Lisp for Data Parallel Programming*. International Conference on SuperComputing, 1987. TMC Tech. Report PL87-6.

[64] Marino, J. Succi, G. and Colla, G. CM2 as an Active Memory to Implement Declarative Languages. *Journal of Programming Language*, 1(2):127–143, June 1993.

[65] Thinking Machines Corp., Cambridge, MA. *Connection Machine CM-5 Technical Summary*, January 1993.

[66] Thinking Machines Corporation. **Lisp Reference Manual*, 1988.

[67] Thinking Machines Corporation, Boston, Massachusetts. *Introduction to PARIS*, 1990.

[68] Turner, D. *An Overview of MIRANDA*. ACM SIGPLAN Notices, New York, Dec. 1986.

[69] Valiant, L. G. Bulk-Synchronous Parallel Computers. Technical Report TR-08-89, Center for Research in Computing Technology, Harvard University, Cambridge, MA, April 1989.

[70] Yonezawa, A. and Tokoro, M., editors. *Object-oriented concurrent programming.* Computer Systems Series. MIT Press, Cambridge, MA, 1987.

[71] Takada, T. Yonezawa, A., Shibayama, E. and Honda, Y. Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. In *Object-Oriented Concurrent Programming*, Computer Systems Series, pages 55–90. MIT Press, Cambridge, MA, 1987.

[72] Yuasa, T. *TUPLE - An Extension of KCL for Massively Parallel SIMD Architecture.* Toyohashi University of Technology, Toyoyashi 441, Japan, draft of 2nd version, 1992. available from author.